What will the following code print?

```
List<String> ls = Arrays.asList("Tom Cruise", "Tom Hart", "Tom Hanks", "Tom Brady");
Predicate<String> p = str->{
    System.out.println("Looking...");
    return str.indexOf("Tom") > -1;
};
boolean flag = ls.stream().filter(str->str.length()>8).allMatch(p);
```

You answered correctly

You had to select 1 option

- It will not print anything.
- It will not print anything but will throw an exception at run time.
- Looking...
  Looking...
  Looking...
  <exception stack trace>
- **Looking...**
  Looking...
  Looking...
- Looking...
  Looking...
  Looking...
  Looking...
- Looking...

Explanation

Remember that `filter` is an intermediate operation, which means it will not execute until a terminal operation is invoked on the stream.
`allMatch` is a short circuiting terminal operation. Thus, when `allMatch` is invoked, the `filter` method will be invoked and it will keep only those elements in the stream that satisfy the condition given in the filter i.e. the string must be longer than 8 characters. After this method is done, only three elements will be left in the stream.
When `allMatch` is invoked, the code in predicate will be executed for each element until it finds a mismatch. Thus, `Looking...` will be printed three times.

Add Note

Previous Next Review Discuss

Given:

```
List<Integer> ls = Arrays.asList(11, 11, 22, 33, 33, 55, 66);
```

Which of the following expressions will return true?

You answered correctly

You had to select 2 options

☐ ~~`ls.stream().anyMatch(44);`~~

This will not compile because anyMatch requires a Predicate object as an argument, not an int.

☐ `ls.stream().anyMatch(11);`

This will not compile because anyMatch requires a Predicate object as an argument, not an int.

☑ `ls.stream().distinct().anyMatch(x->x==11);`

`anyMatch(Predicate<? super T> predicate)` returns whether any elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then false is returned and the predicate is not evaluated.
This is a short-circuiting terminal operation.

☐ ~~`ls.stream().distinct().allMatch(11);`~~

This will not compile because allMatch requires a Predicate object as an argument, not an int. You could do `ls.stream().distinct().allMatch(x->x==11);` but it will return false because for allMatch to return true, the given predicate must return true for each element of the stream.

☑ `ls.stream().noneMatch(x->x%11>0);`

`noneMatch` returns true only if none of the elements in the stream satisfy the given Predicate. Here, all the elements are divisible by `11` and `x%11` will be 0 for each element. Therefore, the given Predicate will return `false` for every element, causing `noneMatch` to return `true`.

Add Note

What can be inserted at //2 so that 6 will be printed by the following code?

```
AtomicInteger ai = new AtomicInteger(5);
//2 INSERT CODE HERE
System.out.println(x);
```

You answered incorrectly

You had to select 2 options

- [x] int x = ai.increment();
- [ ] int x = ai.getAndIncrement();

getAndIncrement() is a valid method call and it will increment ai to 6 but it will return the old value i.e. 5.

- [ ] int x = ai + 1;

AtomicInteger is not a wrapper class and so auto unboxing will not happen here and so it will not compile.

- [x] int x = ai.incrementAndGet();
- [ ] int x = ai.addAndGet(1);
- [ ] int x = ai.getAndSet(6);

ai.getAndSet(int) is a valid method call and it will set ai to 6 but it will return the old value i.e. 5.

Add Note

Identify the correct statements regarding the following program?

```
package trywithresources;

import java.io.IOException;

public class Device implements AutoCloseable{
    String header = null;
    public Device(String name) throws IOException{
        header = name;
        if("D2".equals(name)) throw new IOException("Unknown");
        System.out.println(header + " Opened");
    }

    public String read() throws IOException{
        return "";
    }

    public void close(){
        System.out.println("Closing device "+header);
        throw new RuntimeException("RTE while closing "+header);
    }

    public static void main(String[] args) throws Exception {
        try(Device d1 = new Device("D1");
            Device d2 = new Device("D2")){
            throw new Exception("test");
        }
    }

}
```

You answered incorrectly
You had to select 1 option

- It will end up with an Exception containing message "test".
- It will end up with a RuntimeException containing message "RTE while closing D1"
- It will end up with an IOException containing message "Unknown".
- It will end up with a RuntimeException containing message "RTE while closing D1" and a suppressed IOException containing message "Unknown".
- It will end up with an IOException containing message "Unknown" and a suppressed RuntimeException containing message "RTE while closing D1".

Explanation

The following output obtained after running the program explains what happens:

```
D1 Opened
Closing device D1
Exception in thread "main" java.io.IOException: Unknown
    at trywithresources.Device.<init>(Device.java:9)
    at trywithresources.Device.main(Device.java:24)
    Suppressed: java.lang.RuntimeException: RTE while closing D1
        at trywithresources.Device.close(Device.java:19)
        at trywithresources.Device.main(Device.java:26)
Java Result: 1
```

Device D1 is created successfully but an IOException is thrown while creating Device D2. Thus, the control never enters the try block and `throw new Exception("test")` is never executed. Since one resource was created, its close method will be called (which prints `Closing device D1`). Any exception that is thrown while closing a resource is added to the list of suppressed exceptions of the exception thrown while opening a resource (or thrown from the try block.)

Add Note

Which of the following attributes are supported by `BasicFileAttributeView`?

Left Unanswered

You had to select 3 options

- [ ] archive

This is supported by DosFileAttributeView.

- [ ] isDirectory
- [ ] size
- [ ] system

This is supported by DosFileAttributeView.

- [ ] creationTime

Explanation

The following attributes are supported by BasicFileAttributeView:

Name	Type
"lastModifiedTime"	FileTime
"lastAccessTime"	FileTime
"creationTime"	FileTime
"size"	Long
"isRegularFile"	Boolean
"isDirectory"	Boolean
"isSymbolicLink"	Boolean
"isOther"	Boolean
"fileKey"	Object

Attributes supported by DosFileAttributeView (which extends BasicFileAttributeView) are:

Name	Type
readonly	Boolean
hidden	Boolean
system	Boolean
archive	Boolean

Add Note

Given that Book is a valid class with appropriate constructor and getTitle and getPrice methods that return a String and a Double respectively, what can be inserted at //1 and //2 so that it will print the price of all the books having a title that starts with "A"?

```
List<Book> books = Arrays.asList(
        new Book("Atlas Shrugged", 10.0),
        new Book("Freedom at Midnight", 5.0),
        new Book("Gone with the wind", 5.0)
);

Map<String, Double> bookMap = //1 INSERT CODE HERE
//2 INSERT CODE HERE
bookMap.forEach(func);
```

You answered correctly

You had to select 1 option

(•)
```
books.stream().collect(Collectors.toMap((b->b.getTitle()), b->b.getPrice()));
and
BiConsumer<String, Double> func = (a, b)->{
            if(a.startsWith("A")){
                System.out.println(b);
            }
        };
```

1. The first line generates a `Map<String, Double>` from the List using Stream's collect method. The `Collectors.toMap` method uses two functions to get two values from each element of the stream. The value returned by the first function is used as a key and the value returned by the second function is used as a value to build the resulting Map.

2. The `forEach` method of a Map requires a `BiConsumer`. This function is invoked for each entry, that is each key-value pair, in the map. The first argument of this function is the key and the second is the value.

( )
```
books.stream().toMap((b->b.getTitle()), b->b.getPrice()));
and
BiConsumer<String, Double> func = (a, b)->{
            if(a.startsWith("A")){
                System.out.println(b);
            }
        };
```

toMap is not a valid method in Stream.

( )
```
books.stream().toMap((b->b.getTitle()), b->b.getPrice()));
and
BiConsumer<Map.Entry> func = (b)->{
            if(b.getKey().startsWith("A")){
                System.out.println(b.getValue());
            }
        };
```

1. toMap is not a valid method in Stream.
2. BiConsumer requires two generic types and two arguments.

( )
```
books.stream().collect(Collectors.toMap((b->b.getTitle()), b->b.getPrice()));
and
Consumer<Map.Entry<String, Double>> func = (e)->{
            if(e.getKey().startsWith("A")){
                System.out.println(e.getValue());
            }
    };
```

The implementation of Consumer is technically correct. However, the forEach method requires a BiConsumer.

Add Note

Identify the correct statements about the following code:

```
import java.util.*;
class Account {
    private String id;
    public Account(String id){ this.id = id; }
    //accessors not shown
}
public class BankAccount extends Account{
    private double balance;
    public BankAccount(String id, double balance){ super(id); this.balance = balance;}

    //accessors not shown

    public static void main(String[] args) {
        Map<String, Account> myAccts = new HashMap<>();
        myAccts.put("111", new Account("111"));
        myAccts.put("222", new BankAccount("111", 200.0));

        BiFunction<String, Account, Account> bif =
  (a1, a2)-> a2 instanceof BankAccount?new BankAccount(a1, 300.0):new Account(a1); //1

        myAccts.computeIfPresent("222", bif);//2
        BankAccount ba = (BankAccount) myAccts.get("222");
        System.out.println(ba.getBalance());
    }
}
```

Left Unanswered

You had to select 1 option

- It will not compile due to code at //1.
- It will not compile due to code at //2.
- It will print 200.0
- It will print 300.0

Since myAccts map does contain a key "222", `computeIfPresent` method will execute the function and replace the existing value associated with the given key in the map with the new value returned by the function.

The given function returns a new BankAccount object with a balance of 300.0.

Explanation

Here are a few points that you should know about `java.util.BiFunction` -
1. It is a function that accepts two arguments and produces a result.
2. The types of the arguments and the return value can all be different.

You also need to know about the three flavors of compute methods of Map:
1. `public V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). For example, to either create or append a String msg to a value mapping: `map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))`
(Method merge() is often simpler to use for such purposes.)
If the function returns null, the mapping is removed (or remains absent if initially absent). If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.
Parameters:
`key` - key with which the specified value is to be associated
`remappingFunction` - the function to compute a value
Returns:
the new value associated with the specified key, or null if none

2. `public V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`
If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null. If the function returns null no mapping is recorded. If the function itself throws an (unchecked) exception, the exception is rethrown, and no mapping is recorded. The most common usage is to construct a new object serving as an initial mapped value or memorized result, as in:

```
map.computeIfAbsent(key, k -> new Value(f(k)));
```

Or to implement a multi-value map, Map<K,Collection<V>>, supporting multiple values per key:

```
map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);
```

Parameters:
`key` - key with which the specified value is to be associated
`mappingFunction` - the function to compute a value
Returns:
the current (existing or computed) value associated with the specified key, or null if the computed value is null

Given that daylight Savings Time starts on March 8th at 2 AM in US/Eastern time zone. (As a result, 2 AM becomes 3 AM.), what will the following code print?

```
LocalDateTime ld1 = LocalDateTime.of(2015, Month.MARCH, 8, 2, 0);
ZonedDateTime zd1 = ZonedDateTime.of(ld1, ZoneId.of("US/Eastern"));
LocalDateTime ld2 = LocalDateTime.of(2015, Month.MARCH, 8, 3, 0);
ZonedDateTime zd2 = ZonedDateTime.of(ld2, ZoneId.of("US/Eastern"));
long x = ChronoUnit.HOURS.between(zd1, zd2);
System.out.println(x);
```

You answered incorrectly

You had to select 1 option

- (•) 1
- ( ) -1
- ( ) 0
- ( ) 2
- ( ) -2
- ( ) It will not compile.

Explanation

Think of it as follows -
The time difference between two dates is simply the amount of time you need to go from date 1 to date 2.
So if you want to go from 2AM to 3AM, how many hours do you need? On a regular day, you need 1 hour. That is, if you add 1 hour to 2AM, you will get 3AM. However, as given in the problem statement, at the time of DST change, 2 AM becomes 3AM. That means, even though your local date time is 2 AM, your ZonedDateTime is actually 3AM. Therefore, you are already at 3AM, which means, there is no time difference between 2 AM and 3 AM. The answer is, therefore, 0.

Add Note

Given that Book is a valid class with appropriate constructor and getTitle and getPrice methods that return a String and a Double respectively, what will the following code print?

```
List<Book> books = Arrays.asList(
        new Book("Gone with the wind", 5.0),
        new Book("Gone with the wind", 10.0),
        new Book("Atlas Shrugged", 15.0)
);
books.stream().collect(Collectors.toMap((b->b.getTitle()), b->b.getPrice()))
                      .forEach((a, b)->System.out.println(a+" "+b));
```

You answered correctly
You had to select 1 option

- Gone with the wind 5.0
  Atlas Shrugged 15.0
- Gone with the wind 10.0
  Atlas Shrugged 15.0
- Gone with the wind 5.0
  Gone with the wind 10.0
  Atlas Shrugged 15.0
- (selected) It will throw an exception at run time.

The Collector created by `Collectors.toMap` throws java.lang.IllegalStateException if an attempt is made to store a key that already exists in the Map.
If you want to collect items in a Map and if you expect duplicate entries in the source, you should use `Collectors.toMap(Function, Function, BinaryOperator)` method. The third parameter is used to merge the duplicate entries to produce one entry. For example, in this case, you can do:
`Collectors.toMap(b->b.getTitle(), b->b.getPrice(), (v1, v2)->v1+v2)` This Collector will sum the values of the entries that have the same key. Therefore, it will print :

```
Gone with the wind 15.0
Atlas Shrugged 15.0
```

Add Note

What will the following code print?

```
Stream<String> ss = Stream.of("a", "b", "c");
String str = ss.collect(Collectors.joining(",", "-", "+"));
System.out.println(str);
```

Left Unanswered

You had to select 1 option

- -a+,-b+,-c+
- a,-+b,-+c
- -a,b,c+

`Collectors.joining(",", "-", "+")` returns a Collector that joins all the Strings in the given Stream separated by comma and then prefixes the resulting String with "-" and suffixes the String with "+".

- It will throw an exception at run time.

Explanation

The following JavaDoc description of the joining method is helpful:

```
public static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)
```

Returns a Collector that concatenates the input elements, separated by the specified delimiter, with the specified prefix and suffix, in encounter order.

Parameters:

delimiter - the delimiter to be used between each element prefix - the sequence of characters to be used at the beginning of the joined result suffix - the sequence of characters to be used at the end of the joined result

Returns:

A Collector which concatenates CharSequence elements, separated by the specified delimiter, in encounter order

Add Note

Previous Next Review Discuss

Consider the following program that computes the sum of all integers in a given array of integers:

```
class ComplicatedTask extends RecursiveTask<Integer>{
    int[] ia; int from; int to;
    static final int THRESHOLD = 3;
    public ComplicatedTask(int[] ia, int from, int to){
        this.ia = ia;
        this.from = from;
        this.to = to;
    }
    protected void compute() {
        int sum = 0;
        if(from + THRESHOLD > to){
            for(int i = from; i<=to; i++){
                sum = sum+ia[i];
            }
        }
        else{
            int mid = (from+to)/2;
            ComplicatedTask newtask1 = new ComplicatedTask(ia, from, mid);
            ComplicatedTask newtask2 = new ComplicatedTask(ia, mid+1, to);
            newtask2.fork();
            newtask1.compute();
            newtask2.join();
        }
    }
}
```

What changes must be done together so that it will work as expected?

You answered correctly

You had to select 2 options

- [ ] ~~The class must be made to extend RecursiveAction instead of RecursiveTask.~~

RecursiveAction is used when a task does not return a value. In this case, we want it to return an integer value so it should extend `RecursiveTask<Integer>`.

- [x] The `compute` method must be changed to return a value.
- [ ] A new task must be forked from the if block instead of the else block.
- [x] The values returned by the `newtask1` and `newtask2` should be added and returned.
- [ ] The class must override `fork()` to return an Integer value.

Explanation

The following is the complete program that illustrates how this program can be fixed:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class ComplicatedTask extends RecursiveTask<Integer>{
    int[] ia; int from; int to;
    static final int THRESHOLD = 3;
    public ComplicatedTask(int[] ia, int from, int to){
        this.ia = ia;
        this.from = from;
        this.to = to;
    }
    public int transform(int t){
      //this can be a CPU intensive operation that
      //transforms t and returns the value
      //For now, just return t
        return t;
    }
    protected Integer compute() {
        int sum = 0;
        if(from + THRESHOLD > to){
            for(int i = from; i<=to; i++){
                sum = sum+transform(ia[i]);
            }
        }
        else{

            int mid = (from+to)/2;
            ComplicatedTask newtask1 = new ComplicatedTask(ia, from, mid);
            ComplicatedTask newtask2 = new ComplicatedTask(ia, mid+1, to);
```

Given:

```
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentHashMap;
public class Cache {

    static ConcurrentHashMap<String, Object> chm = new ConcurrentHashMap<String, Object>();

    public static void main(String[] args) {
        chm.put("a", "aaa");
        chm.put("b", "bbb");
        chm.put("c", "ccc");

        new Thread(){
            public void run(){
                Iterator<Entry<String, Object>> it = Cache.chm.entrySet().iterator();
                while(it.hasNext()){
                    Entry<String, Object> en = it.next();
                    if(en.getKey().equals("a") || en.getKey().equals("b")){
                        it.remove();
                    }
                }
            }
        }.start();

        new Thread(){
            public void run(){
                Iterator<Entry<String, Object>> it = Cache.chm.entrySet().iterator();
                while(it.hasNext()){
                    Entry<String, Object> en = it.next();
                    System.out.print(en.getKey()+", ");
                }
            }
        }.start();
    }
}
```

Which of the following are possible outputs when the above program is run?

Left Unanswered

You had to select 1 option

- It may print any combination of the keys.
- It may print any combination except: c,
- It may print any combination except: a, or b, or a, b, or b, a

This is correct because the order of iteration is not known and so the thread that removes "a" and "b", may remove them in any order. Thus, the iterator thread may or may not see "a" and/or "b" through its Iterator.
However, "c" is never removed from the map and so c, will always be printed.

- It may print any combination except: b, c,
- It may print any combination except: a, b,

Explanation

An important thing to know about the Iterators retrieved from a ConcurrentHashMap is that they are backed by that ConcurrentHashMap, which means any operations done on the ConcurrentHashMap instance may be reflected in the Iterator.

Thus, in this case, the thread that is iterating through the entries may or may not see the entries removed from the map by another thread. The following is what JavaDoc API description says about ConcurrentHashMap:

Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset. For aggregate operations such as putAll and clear, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do not throw ConcurrentModificationException. However, iterators are designed to be used by only one thread at a time.

and the following is the behaviour description of the EntrySet retrieved from a ConcurrentHashMap instance using the entrySet() method:

entrySet() returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

The view's iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.

Add Note

Previous Next Review Discuss*

Given:

```
public class ItemProcessor implements Runnable{  //LINE 1
    CyclicBarrier cb;
    public ItemProcessor(CyclicBarrier cb){
        this.cb = cb;
    }
    public void run(){
       System.out.println("processed");
        try {
            cb.await();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

public class Merger implements Runnable{ //LINE 2
    public void run(){
        System.out.println("Value Merged");
    }
}
```

What should be inserted in the following code such that run method of Merger will be executed only after the thread started at 4 and the main thread have both invoked await?

```
    public static void main(String[] args) throws Exception{
        Merger m = new Merger();

        //LINE 3

        ItemProcessor ip = new ItemProcessor(cb);
        ip.start(); //LINE 4
        cb.await();
    }
```

You answered correctly

You had to select 1 option

- Make `ItemProcessor` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- **Make `ItemProcessor` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3**

1. ItemProcessor needs to extend Thread otherwise ip.start() will not compile.
2. Since there are a total two threads that are calling cb.await (one is the ItemProcessor thread and another one is the main thread), you need to create a CyclicBarrier with number of parties parameter as 2. If you specify the number of parties parameter as 1, Merger's run will be invoke as soon as the any thread invokes await but that is not what the problem statement wants.

- Make `Merger` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- Make `Merger` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3
- Just add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- Just add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3

Explanation

Briefly, a CyclicBarrier allows multiple threads to run independently but wait at one point until all of the coordinating threads arrive at that point. Once all the threads arrive at that point, all the threads can then proceed. It is like multiple cyclists taking different routes to reach a particular junction. They may arrive at different times but they will wait there until everyone arrives. Once everyone is there, they can go on futher independent of each other.

It is important for you to understand how CyclicBarrier works and is used. Make sure you understand both the constructors of CyclicBarrier.

Here are a few links that explain it quite well:
http://examples.javacodegeeks.com/core-java/util/concurrent/cyclicbarrier/java-util-concurrent-cyclicbarrier-example/
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html
http://www.sourcetricks.com/2014/09/java-cyclicbarrier.html#.VZIPPvlVif8

Add Note

Previous Next Review Discuss

Given:

```
public class ItemProcessor implements Runnable{  //LINE 1
    CyclicBarrier cb;
    public ItemProcessor(CyclicBarrier cb){
        this.cb = cb;
    }
    public void run(){
       System.out.println("processed");
        try {
            cb.await();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

public class Merger implements Runnable{ //LINE 2
    public void run(){
        System.out.println("Value Merged");
    }
}
```

What should be inserted in the following code such that run method of Merger will be executed only after the thread started at 4 and the main thread have both invoked await?

```
    public static void main(String[] args) throws Exception{
        Merger m = new Merger();

        //LINE 3

        ItemProcessor ip = new ItemProcessor(cb);
        ip.start(); //LINE 4
        cb.await();
    }
```

Left Unanswered

You had to select 1 option

- Make `ItemProcessor` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- **Make `ItemProcessor` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3**

1. ItemProcessor needs to extend Thread otherwise ip.start() will not compile.
2. Since there are a total two threads that are calling cb.await (one is the ItemProcessor thread and another one is the main thread), you need to create a CyclicBarrier with number of parties parameter as 2. If you specify the number of parties parameter as 1, Merger's run will be invoke as soon as the any thread invokes await but that is not what the problem statement wants.

- Make `Merger` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- Make `Merger` extend `Thread` instead of implementing Runnable and add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3
- Just add `CyclicBarrier cb = new CyclicBarrier(1, m);` to //LINE 3
- Just add `CyclicBarrier cb = new CyclicBarrier(2, m);` to //LINE 3

Explanation

Briefly, a CyclicBarrier allows multiple threads to run independently but wait at one point until all of the coordinating threads arrive at that point. Once all the threads arrive at that point, all the threads can then proceed. It is like multiple cyclists taking different routes to reach a particular junction. They may arrive at different times but they will wait there until everyone arrives. Once everyone is there, they can go on futher independent of each other.

It is important for you to understand how CyclicBarrier works and is used. Make sure you understand both the constructors of CyclicBarrier.

Here are a few links that explain it quite well:
http://examples.javacodegeeks.com/core-java/util/concurrent/cyclicbarrier/java-util-concurrent-cyclicbarrier-example/
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html
http://www.sourcetricks.com/2014/09/java-cyclicbarrier.html#.VZIPPvlVif8

Add Note

Identify correct statements about Java Date and Time API.

Left Unanswered

You had to select 1 option

- [x] Classes used to represent dates and times in java.time package are thread safe.

All classess in java.time package such as classes for date, time, date and time combined, time zones, instants, duration, and clocks are immutable and thread-safe.

- [ ] Time zone is an integral part of all classes in java.time package.

Only classes that are explicitly designed to work with time zones (such as ZonedDateTime, OffsetDateTime, and OffsetDate) incorporate the time zone. All other classes such as Instant , LocalDate, LocalTime, Period, and Duration do not have any time zone.

- [ ] Period class is ideal for usage as a time stamp.

java.time.Instant class represents the start of a nanosecond on the timeline. This class is ideal for generating a time stamp to represent machine time.

- [ ] Duration class handles day light saving time changes.

Daylight Saving Time is related to Time zones. It has nothing to do with Duration or Period.

A Duration object is measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes. A Duration can have a negative value, if it is created with an end point that occurs before the start point.

- [ ] You can create instances of Instant using one of the several constructors that take date, time, or date time values.

Instant class does not have public constructors. You create Instant objects by using it static helper methods such as `Instant.now()` or by using existing Instant objects, for example: `Instant newInstant = oldInstant.plusHours(1);`

Add Note

What will the following code print when compiled and run?

```
import java.time.*;
import java.time.format.*;
public class DateTest{
  public static void main(String[] args){ //1
        LocalDateTime greatDay = LocalDateTime.parse("2015-01-01");//2
        String greatDayStr = greatDay.format(DateTimeFormatter.ISO_DATE_TIME); //3
        System.out.println(greatDayStr);//4
   }
}
```

You answered incorrectly

You had to select 1 option

- (•) //1 will not compile because of lack of throws clause.

Operations in the new date/time related classes throw `java.time.DateTimeException`, which extends from `RuntimeException`. Therefore, this exception is not required to be caught or declared in the throws clause.

- ( ) //2 will not compile because of invalid date string.

The given date string does not contain a time component and so it cannot be parsed by LocalDateTime. However, this is a run time issue and not a compile time one.

- ( ) //2 will throw an exception at run time.

It will throw a DateTimeException because it doesn't have time component.

```
Exception in thread "main" java.time.format.DateTimeParseException: Text '2015-01-01' could not be parsed at index 10.
```

A String such as 2015-01-01T17:13:50 would have worked.

- ( ) It will print 2015-01-01T00:00:00

- ( ) It will print null.

Add Note

What will the following code print when compiled and run?

```
import java.time.*;
import java.time.format.*;
public class DateTest{
  public static void main(String[] args){ //1
        LocalDateTime greatDay = LocalDateTime.parse("2015-01-01");//2
        String greatDayStr = greatDay.format(DateTimeFormatter.ISO_DATE_TIME); //3
        System.out.println(greatDayStr);//4
   }
}
```

You answered incorrectly

You had to select 1 option

- (•) //1 will not compile because of lack of throws clause.

Operations in the new date/time related classes throw `java.time.DateTimeException`, which extends from `RuntimeException`. Therefore, this exception is not required to be caught or declared in the throws clause.

- ( ) //2 will not compile because of invalid date string.

The given date string does not contain a time component and so it cannot be parsed by LocalDateTime. However, this is a run time issue and not a compile time one.

- ( ) //2 will throw an exception at run time.

It will throw a DateTimeException because it doesn't have time component.

```
Exception in thread "main" java.time.format.DateTimeParseException: Text '2015-01-01' could not be parsed at index 10.
```

A String such as 2015-01-01T17:13:50 would have worked.

- ( ) It will print 2015-01-01T00:00:00

- ( ) It will print null.

Add Note

What will the following line of code print?

```
System.out.println(LocalDate.of(2015, Month.JANUARY, 31).format(DateTimeFormatter.ISO_DATE_TIME));
```

You answered incorrectly

You had to select 1 option

- (•) 31 Jan 2015
- ( ) 31 January 2015 00:00:00
- ( ) 2015-01-31
- ( ) 2015-01-31T00:00:00
- ( ) Exception at run time.

Observe that you are creating a `LocalDate` and not a `LocalDateTime`. `LocalDate` doesn't have time component and therefore, you cannot format it with a formatter that expects time component such as `DateTimeFormatter.ISO_DATE_TIME`.
Thus, it will print `java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: HourOfDay` exception message.

If you use `DateTimeFormatter.ISO_DATE`, it will print `2015-01-31`
.

Also, remember that a `LocalDateTime` object can be formatted using a `DateTimeFormatter.ISO_DATE` though.

Add Note

Consider the following code:

```
import java.util.*;
import java.text.*;

public class TestClass
{
    public static void main(String[] args) throws Exception
    {
        double amount = 53000.35;

        Locale jp = new Locale("jp", "JP");

        //1 create formatter here.

        System.out.println(   formatter.format(amount)  );
    }
}
```

How will you create formatter using a factory at //1 so that the output is in Japanese Currency format?

You answered incorrectly
You had to select 2 options

☐ `NumberFormat formatter =  NumberFormat.getCurrencyFormatter(jp);`

☐ `NumberFormat formatter =  new DecimalFormat(jp);`

1. `DecimalFormat` has no constructor that takes a `Locale`.
2. Creating an object using new as done in this option means you are not using a factory. Remember, using a factory to get an object, usually means calling `getInstance()` or `getXXXInstance()` method on a Factory class such as `NumberFormat` or `DateFormat`.

☑ `Format formatter =  NumberFormat.getCurrencyInstance(jp);`

This is valid because `java.text.NumberFormat` extends from `java.text.Format`. The return type of the method `getCurrencyInstance()` is `NumberFormat`.

☐ `NumberFormat formatter =  DecimalFormat.getCurrencyInstance(jp);`

`getCurrencyInstance` is actually defined in `NumberFormat`. However, since `DecimalFormat` extends `NumberFormat`, this is valid.

To format a number in currency format, you should use `getCurrencyInstance()` instead of `getInstance()` or `getNumberInstance()`.

This will print : `JPY 53,000`

☑ `NumberFormat formatter =  NumberFormat.getInstance(jp);`

`getInstance(Locale )` is a valid factory method in `NumberFormat` class but it will not not format the given number as per the currency.

☐ `NumberFormat formatter =  new DecimalFormat("#.00");`

While it is a valid way to create a DecimalFormat object, it is not valid for two reasons:
1. We need a currency formatter and not just a simple numeric formatter.
2. This is not using a factory to create the formatter object.

Explanation

To obtain a `NumberFormat` for a specific locale, including the default locale, call one of `NumberFormat`'s factory methods, such as `getInstance()`. In general, do not call the `DecimalFormat` constructors directly, since the `NumberFormat` factory methods may return subclasses other than `DecimalFormat`. If you need to customize the format object, do something like this:

```
NumberFormat f = NumberFormat.getInstance(loc);
if (f instanceof DecimalFormat) {
     ((DecimalFormat) f).setDecimalSeparatorAlwaysShown(true);
}
```

@pomocnicze : java - Konsola
Na Pulpit 1

Add Note

Previous Next Review Discuss*

Which of the following are valid?

Left Unanswered

You had to select 2 options

- [ ] `List<String> list = new List<>();`

List is an interface. It cannot be instantiated, therefore `new List<>()` is invalid.

- [ ] `ArrayList<String> list = new List<>();`
- [ ] `ArrayList<> list = new ArrayList<>();`
- [ ] `List<String> list = new ArrayList<>();`
- [ ] `List<String> list = new ArrayList<>(10);`

While creating an `ArrayList` you can pass in an integer argument that specifies the initial size of the ArrayList. This doesn't actually insert any element into the list. So `list.getSize()` would still return 0.

Explanation

The following explanation about the diamond operator from Oracle is sufficient for the exam (http://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html):

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

`Map<String, List<String>> myMap = new HashMap<String, List<String>>();` In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

`Map<String, List<String>> myMap = new HashMap<>();` Note that to take advantage of automatic type inference during generic class instantiation, you must specify the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

`Map<String, List<String>> myMap = new HashMap();` // unchecked conversion warningJava SE 7 supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context. For example, the following example does not compile on Java 7 but compiles on Java 8:

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

Note that the diamond often works in method calls; however, it is suggested that you use the diamond primarily for variable declarations.

In comparison, the following example compiles:

```
List<? extends String> list2 = new ArrayList<>();
list.addAll(list2);
```

Add Note

Given:

```
Map<String , List<? extends CharSequence>> stateCitiesMap = new HashMap<String, List<? extends CharSequence>>();
```

Which of the following options correctly achieves the same declaration using type inferencing?

Left Unanswered

You had to select 1 option

- `Map<String , List<? extends CharSequence>> stateCitiesMap = new HashMap<String, List<>>();`

This will not compile because you can't use a diamond operator within a generic type specification i.e. `<String, List<>>` is wrong because of `List<>` being inside of another < and >.

- `Map<String , List<? extends CharSequence>> stateCitiesMap = new HashMap();`

While this is valid code but this does not take advantage of type inferencing and will generate a warning at compile time. To use type inferencing, the diamond operator must be used.

- `Map<String , List<? extends CharSequence>> stateCitiesMap = new HashMap<>();`
- `Map<String , List<? extends CharSequence>> stateCitiesMap = new HashMap<<>>();`

Add Note

What will the following code print?

```
Duration d = Duration.ofHours(25);
System.out.println(d);
Period p = Period.ofDays(1);
System.out.println(p);
```

You answered incorrectly
You had to select 1 option

- ~~PT25H~~
  ~~P1D~~

There are two important things to note here:
1. Duration string starts with PT (because duration is "time" based i.e. hours/minutes/seconds) and Period string starts with just P (because period does not include time. It contains only years, months, and days).

2. Duration does not convert hours into days. i.e. 25 hours will remain as 25 hours instead of 1 day and 1 hour.

- PT1D1H
  P1D
- ~~PT25H~~
  ~~PT1D~~
- PT1D1H
  PT1D

Explanation

The following is from JavaDoc API description of the toString methods of Duration and Period.

`Duration.toString`:

It generates a string representation of the duration object using ISO-8601 seconds based representation, such as PT8H6M12.345S.

The format of the returned string will be PTnHnMnS, where n is the relevant hours, minutes or seconds part of the duration. Any fractional seconds are placed after a decimal point i the seconds section. If a section has a zero value, it is omitted. The hours, minutes and seconds will all have the same sign.
Examples:

```
    "20.345 seconds"                 -- "PT20.345S
    "15 minutes" (15 * 60 seconds)   -- "PT15M"
    "10 hours" (10 * 3600 seconds)   -- "PT10H"
    "2 days" (2 * 86400 seconds)     -- "PT48H"
```

Note that multiples of 24 hours are not output as days to avoid confusion with Period.

`Period.toString`:
Outputs this period as a String, such as P6Y3M1D.
The output will be in the ISO-8601 period format. A zero period will be represented as zero days, 'P0D'.

Add Note

What will the following code print?

```
Duration d = Duration.ofMillis(1100);
System.out.println(d);
d = Duration.ofSeconds(61);
System.out.println(d);
```

You answered incorrectly

You had to select 1 option

- PT1S PT1M1S
- PT1S1m PT1M1S
- PT1.1S PT61S
- PT1.1S PT1M1S

Explanation

This question is based on how the toString method of java.time.Duration works -

It generates a string representation of the duration object using ISO-8601 seconds based representation, such as PT8H6M12.345S.

The format of the returned string will be PTnHnMnS, where n is the relevant hours, minutes or seconds part of the duration. Any fractional seconds are placed after a decimal point i the seconds section. If a section has a zero value, it is omitted. The hours, minutes and seconds will all have the same sign.

Examples:

```
    "20.345 seconds"                 -- "PT20.345S
    "15 minutes" (15 * 60 seconds)   -- "PT15M"
    "10 hours" (10 * 3600 seconds)   -- "PT10H"
    "2 days" (2 * 86400 seconds)     -- "PT48H"
```

Note that multiples of 24 hours are not output as days to avoid confusion with Period.

Add Note

Given: Daylight Savings Time ends on Nov 1 at 2 AM in US/Eastern time zone. As a result, 2 AM becomes 1 AM.What will the following code print ?

```
LocalDateTime ld = LocalDateTime.of(2015, Month.OCTOBER, 31, 10, 0);

ZonedDateTime date = ZonedDateTime.of(ld, ZoneId.of("US/Eastern"));
date = date.plus(Duration.ofDays(1));
System.out.println(date);

date = ZonedDateTime.of(ld, ZoneId.of("US/Eastern"));
date = date.plus(Period.ofDays(1));
System.out.println(date);
```

Left Unanswered

You had to select 1 option

- 2015-11-01T09:00-05:00[US/Eastern] 2015-11-01T09:00-05:00[US/Eastern]
- **2015-11-01T09:00-05:00[US/Eastern] 2015-11-01T10:00-05:00[US/Eastern]**
- 2015-11-01T10:00-05:00[US/Eastern] 2015-11-01T09:00-05:00[US/Eastern]
- 2015-11-01T10:00-05:00[US/Eastern] 2015-11-01T10:00-05:00[US/Eastern]

Explanation

Important thing to remember here is that Period is used to manipulate dates in terms of days, months, and years, while Duration is used to manipulate dates in terms of hours, minutes, and seconds. Therefore, Period doesn't mess with the time component of the date while Duration may change the time component if the date is close to the DST boundary.

Durations and periods differ in their treatment of daylight savings time when added to ZonedDateTime. A Duration will add an exact number of seconds, thus a duration of one day is always exactly 24 hours. By contrast, a Period will add a conceptual day, trying to maintain the local time.

For example, consider adding a period of one day and a duration of one day to 18:00 on the evening before a daylight savings gap. The Period will add the conceptual day and result in a ZonedDateTime at 18:00 the following day. By contrast, the Duration will add exactly 24 hours, resulting in a ZonedDateTime at 19:00 the following day (assuming a one hour DST gap).

Add Note

Which of the following code fragments will you use to create an ExecutorService?

Left Unanswered

You had to select 2 options

- [ ] `ExecutorService`
- [ ] `Executor`
- [ ] `Executors`
- [ ] `.newSingleThreadExecutor();`
- [ ] `.createSingleThreadExecutor();`
- [ ] `.getSingleThreadExecutor();`

Explanation

You need to remember the following points about a few important classes in `java.util.concurrent` package:

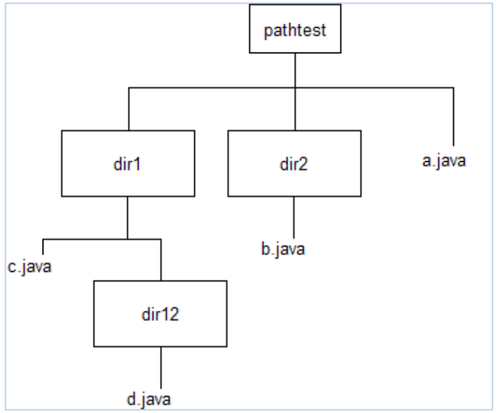
1. `ExecutorService` interface extends `Executor` interface. While `Executor` allows you to execute a `Runnable`, `ExecutorService` allows you to execute a `Callable`.

2. `Executors` is a utility class that provides several static methods to create instances of ExecutorService. All such methods start with `new` e.g. `newSingleThreadExecutor()`.
You should at least remember the following methods: `newFixedThreadPool(int noOfThreads)`, `newSingleThreadExecutor()`, `newCachedThreadPool()`, `newSingleThreadScheduledExecutor()`, `newScheduledThreadPool(int corePoolSize)`.

Add Note

```
import java.nio.file.attribute.*;
import java.io.*;
class MyFileChecker extends SimpleFileVisitor<Path>{
    private final PathMatcher matcher;
    private static int count;

    public MyFileChecker(){
        matcher = FileSystems.getDefault().getPathMatcher("glob:*.java");
    }

    void check(Path p){
        Path name = p.getFileName();
        if(name != null && matcher.matches(name)){
            count++;
        }

    }
    public int getCount(){
        return count;
    }

    public FileVisitResult visitFile(Path p, BasicFileAttributes attr){
        check(p);
        return FileVisitResult.CONTINUE;
    }
}

public class Path2 {
    public static void main(String[] args) throws IOException {
        MyFileChecker mfc = new MyFileChecker();
        Files.walkFileTree(Paths.get("c:\\works\\pathtest"), mfc);
        System.out.println(mfc.getCount());
    }

}
```

What will be the output when this program is run?



You answered incorrectly
You had to select 1 option

- Compilation fails

`Path name = p.getFileName();` seems like a weird call but it is valid. `Path.getFileName()` returns a Path object that contains the file name of the complete path. For example, for `c:\works\pathtest\a.java`, it will return a Path object contains just `"a.java"`.

- 1
- 3
- 4

Note that `Files.walkFileTree` method will cause each subdirectory under the given directory to be travelled. For each file in each directory, the FileVisitor's visitFile will be invoked. This particular visitor simply tries to match the full file name with the given glob pattern. The glob pattern will match only the files ending with .java.

- 0

Add Note

Given:

```
Path p = Paths.get("c:\\temp\\test.txt");
```

Which of the following statements will make the file test.txt hidden?

You answered correctly

You had to select 1 option

- `Files.setAttribute(p, "dos:hidden");`
- **`Files.setAttribute(p, "dos:hidden", true);`**
- `Files.setAttribute(p.getFile(), "dos:hidden");`

Note that there is a `getFileName()` and a `getFileSystem()` method in `Path` class but no `getFile()`. `toFile()` is the right method to get a `File` object from a `Path` object. However, it is still not a correct option because `Files.setAttribute` works on a `Path` object and not a `File` object.

- `Files.setAttribute(p.toFile(), "dos:hidden", true);`

Explanation

The following API description of the setAttribute method explains all that you need to know about this method:

```
public static Path setAttribute(Path path,
                String attribute,
                Object value,
                LinkOption... options)
                        throws IOException
```

Sets the value of a file attribute.
The attribute parameter identifies the attribute to be set and takes the form: `[view-name:]attribute-name`
where square brackets [...] delineate an optional component and the character ':' stands for itself.

`view-name` is the name of a `FileAttributeView` that identifies a set of file attributes. If not specified then it defaults to "basic", the name of the file attribute view that identifies the basic set of file attributes common to many file systems. attribute-name is the name of the attribute within the set.

The options array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is set. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Usage Example:

Suppose we want to set the DOS "hidden" attribute:

```
Path path = ...
Files.setAttribute(path, "dos:hidden", true);
```

Add Note

Q 44 of 60 QID : enthuware.ocpjp.v8.2.1219

Hide Section/Toughness Mark

Which of the following are valid enum values defined in java.nio.file.FileVisitResult?

You answered correctly
You had to select 2 options

- [x] SKIP_SIBLINGS
- [ ] CONTINUE_SIBLINGS

There is no such value.

- [x] SKIP_SUBTREE
- [ ] SKIP_TREE

There is no such value.

- [ ] SKIP

There is no such value.

Explanation

java.nio.file.FileVisitResult defines the following four enum constants -

CONTINUE
Continue.

SKIP_SIBLINGS
Continue without visiting the siblings of this file or directory.

SKIP_SUBTREE
Continue without visiting the entries in this directory.

TERMINATE
Terminate.

Add Note

Given :

```
interface Process{
    public void process(int a, int b);
}

public class Data{
    int value;
    Data(int value){
        this.value = value;
    }
}
```

and the following code fragments:

```
public void processList(ArrayList<Data> dataList, Process p){
   for(Data d: dataList){
        p.process(d.value, d.value);
   }
}

....
        ArrayList<Data> al = new ArrayList<Data>();
        al.add(new Data(1));al.add(new Data(2));al.add(new Data(3));

        //INSERT METHOD CALL HERE
```

Which of the following options can be inserted above so that it will print 1 4 9?

You answered incorrectly

You had to select 3 options

☐ ~~processList(al, a, b->System.out.println(a*b));~~

Observe that without the brackets over a, b, it would imply that you are trying to pass 3 arguments to processList method - a, b, and b->System.out.println(a*b), which is incorrect. You actually want to pass only two arguments - a and the lambda expression. Therefore, whenever the method of a functional interface takes more than one parameter, you need to put the arguments within brackets.

If the method of a functional interface takes one parameter, you can omit the brackets. For example, `x -> expression` and `(x) -> expression` are equivalent.

If the method of a functional interface takes no parameter, you must write empty brackets. For example, `( ) -> expression`

☐ ~~processList(al, (int a, int b)->System.out.println(a*b));~~

☑ `processList(al, (int a, int b)->System.out.println(a*b); );`

When your method body comprises only a single expression, you must omit the semi-colon.

☑ `processList(al, (a, b)->System.out.println(a*b));`

It is ok to omit the parameter types in case of a functional interface because the compiler can determine the type of the parameters by looking at the interface method.

☑ `processList(al, (a, b) ->{  System.out.println(a*b); } );`

If you enclose your method body within curly braces, you must write complete lines of code including the semi-colon.
FYI, if the method is supposed to return a value, then you must include a return statement just like you do in a regular method if you are using the curly braces syntax.

Explanation

There is a simple trick to identify invalid lambda constructs. When you write a lambda expression for a functional interface, you are essentially providing an implementation of the method declared in that interface but in a very concise manner. Therefore, the lambda expression code that you write must contain all the pieces of the regular method code except the ones that the compiler can easily figure out on its own such as the parameter types, return keyword, and brackets. So, in a lambda expression, just check that all the information is there and that the expression follows the basic syntax -

```
(parameter list) OR single_variable_without_type -> { regular lines of code } OR just_an_expression_without_semicolon
```

For a complete discussion on this topic please see this short tutorial - http://enthuware.com/index.php/home/115

Add Note

Given:

```
//assume appropriate imports
public class Calculator{
    public static void main(String[] args) {
        double principle = 100;
        int interestrate = 5;
        double amount = compute(principle, x->x*interestrate);
    }

    INSERT CODE HERE
}
```

Which of the following methods can be inserted in the above code so that it will compile and run without any error/exception?

Left Unanswered

You had to select 2 options

☐
```
public static double compute(double base, Function<Integer, Integer > func){
    return func.apply((int)base);
}
```

☐
```
public static double compute(double base, Function<Integer, Double> func){
    return func.apply((int)base);
}
```
The method definition is fine. But the usage of this method i.e. `compute(principle, x->x*interestrate);` will not compile because the type of the expression that makes up the body of the function i.e. `x*interestrate` is int, while the expected return type of the Function is Double. int cannot be boxed to a Double.
You could do this though: `double amount = compute(principle, x->new Double(x*interestrate));`

☐
```
public static double compute(double base, Function<Double, Integer> func){
    return func.apply(base);
}
```
`Function<Double, Integer>` implies that your argument type is `Double` and return type is `Integer`. However, the return type of the lambda expression `x->x*interestrate` is Double because you are passing base, which is a double, as the argument to this function..

☐
```
public static double compute(double base, Function<Double, Double> func){
    return func.apply(base);
}
```

☐
```
public static double compute(double base, Function<Integer, Double> func){
    return func.apply(base);
}
```
`Function<Integer, Double>` implies that your argument type is Integer and return type is Double. However, you are passing `base`, which is double, to this function. You cannot pass a `double` where an `int` or `Integer` is required without explicit cast due to potential loss of precision.

My Note : do przeanalizowania Wt, cze 14, '16 23:55:06

Edit Note

Q 39 of 60 QID : enthuware.ocpjp.v8.2.1878 Hide Section/Toughness Ma

Given:

```
//assume appropriate imports
public class Calculator{
   public static void main(String[] args) {
       double principle = 100;
       int interestrate = 5;
       double amount = compute(principle, x->x*interestrate);
   }

   INSERT CODE HERE
```

Which of the following methods can be inserted in the above code so that it will compile and run without any error/exception?

Left Unanswered
You had to select 2 options

```
public static double compute(double base, Function<Integer, Integer > func){
    return func.apply((int)base);
}
```

```
public static double compute(double base, Function<Integer, Double> func){
    return func.apply((int)base);
}
```

The method definition is fine. But the usage of this method i.e. `compute(principle, x->x*interestrate);` will not compile because the type of the expression that makes up the body of the function i.e. `x*interestrate` is int, while the expected return type of the Function is Double. int cannot be boxed to a Double.
You could do this though: `double amount = compute(principle, x->new Double(x*interestrate));`

```
public static double compute(double base, Function<Double, Integer> func){
    return func.apply(base);
}
```

`Function<Double, Integer>` implies that your argument type is `Double` and return type is `Integer`. However, the return type of the lambda expression `x->x*interestrate` is Double because you are passing base, which is a double, as the argument to this function..

```
public static double compute(double base, Function<Double, Double> func){
    return func.apply(base);
}
```

```
public static double compute(double base, Function<Integer, Double> func){
    return func.apply(base);
}
```

`Function<Integer, Double>` implies that your argument type is Integer and return type is Double. However, you are passing `base`, which is double, to this function. You cannot pass a `double` where an `int` or `Integer` is required without explicit cast due to potential loss of precision.

My Note : do przeanalizowania Wt, cze 14, '16 23:55:06

Edit Note

Given:
```
String[] p = {"1", "2", "3" };
```
Which of the following lines of code is/are valid?

You answered incorrectly
You had to select 1 option

- ( ) `List<?> list2 = new ArrayList<Integer>(Arrays.asList(p));`
- ( ) `List<Integer> list2 = new ArrayList<Integer>(Arrays.asList(p));`
- ( ) `List<Integer> list2 = new ArrayList<>(Arrays.asList(p));`
- (•) `List<?> list2 = new ArrayList<>(Arrays.asList(p));`

Here, `list2` is a list of any thing. You cannot add any thing to it and you can only retrieve Objects from it:
```
list2.add(new Object()); list2.add("aaa"); //both will not compile.
Object obj = list2.get(0); //Valid
String str = list2.get(0); //will not compile.
```
Note that you can add `null` to it though i.e. `list2.add(null);` is valid.

Explanation

`Arrays.asList(T[] )` returns an object of type `ArrayList<T>`. In this case, it will be `ArrayList<String>`. Therefore, elements in this list cannot be added to a List of Integers.

Add Note

Which of the following code fragments is/are appropriate usage(s) of generics?

You answered incorrectly
You had to select 2 options

- [x] `Map<String, List<String>> myMap = new HashMap();`

Although this is valid code but it is not a good usage of generics because when you do `new HashMap()`, you create a raw HashMap and it will produce a warning at compile time.

Note that to take advantage of automatic type inference during generic class instantiation, you must specify the diamond operator. i.e. You must do `new HashMap<>()` in this case.

- [ ]
```
List<String> list = new ArrayList<>();
list.add("A");
list.addAll(new ArrayList<>());
```

Note that this code will work in Java 8 but will not work in Java 7 and older versions. Java SE 7 supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context.
In this option, the last line will not compile because `list.addAll()` method expects `Collection<? extends String>` and so the compiler cannot infer the generic type for ArrayList.

- [x]
```
List<String> list = new ArrayList<>();
list.add("A");
List<? extends String> list2 = new ArrayList<>();
list.addAll(list2);
```

- [ ] `List<> list = new ArrayList<String>();`

The diamond operator is used at the place of creation of the object and not at the place of variable type declaration. So it should be: `List<String> list = new ArrayList<>();`

Add Note

Identify valid methods:

Assume that Shape is a valid non-final class.

You answered incorrectly
You had to select 2 options

```
public List<Shape> m3(ArrayList<? extends Shape> strList){
    List<? extends Shape> list = new ArrayList<>();
    list.addAll(strList);
    return list;
}
```

strList and list both are Lists of some class that extends from Shape. However, the compiler does not know which class(es). Therefore, you cannot add contents of strList to list or vice-versa.

```
public  List<? extends Shape> m4(List<? extends Shape> strList){
    List<Shape> list = new ArrayList<>();
    list.add(new Shape());
    list.addAll(strList);
    return list;
}
```

Here, list is a List of Shapes and strList is a List of some class that extends from Shape. Any class that extends from Shape IS-A Shape, therefore, you can add elements in strList to list.

```
public void m5(ArrayList<? extends Shape> strList){
    List<Shape> list = new ArrayList<>();
    list.add(new Shape());
    list.addAll(strList);
}
```

Same as above.

```
public void m6(ArrayList<Shape> strList){
    List<? extends Shape> list = new ArrayList<>();
    list.add(new Shape());
    strList.addAll(list);
}
```

list is a List of some class that extends from Shape. It may not necessarily be Shape. Therefore, you cannot add a Shape to list.
But you can add all the elements of list to strList.

Add Note

Given :

glob pattern: `glob:?{pdf,rtf}`
Actual name of the files in a directory (Separated by comma, not including the space):
`.rtf, a.pdf, ab.rtf, ab.pdf, pdf`

Which files will be captured by the glob pattern?

Left Unanswered
You had to select 1 option

- [x] `.rtf`

Remember that ? matches exactly one character in a glob pattern (as opposed to zero or one in a regex). Therefore, `?{pdf,rtf}` will match only those file names that have exactly four characters and the last three characters have to be `pdf` or `rtf`.

- [ ] `a.pdf`
- [ ] `ab.rtf`
- [ ] `ab.pdf`
- [ ] `pdf`

Since ? matches exactly 1 character in a glob pattern, this doesn't satisfy glob pattern `?pdf`. But the following would print true:

```
Pattern p = Pattern.compile(".?pdf");  //Notice that .? matches 0 or 1 character
Matcher m = p.matcher("pdf");
System.out.println(m.matches()); //prints true.
Matcher m = p.matcher(".pdf");
System.out.println(m.matches()); //prints true.
```

Add Note

Given:

```
public class Student {

    public static enum Grade{ A, B , C, D, F}

    private String name;
    private Grade grade;
    public Student(String name, Grade grade){
        this.name = name;
        this.grade = grade;
    }
    public String toString(){
        return name+":"+grade;
    }
    //getters and setters not shown
}
```

What can be inserted in the code below so that it will print:
{C=[S3], A=[S1, S2]}

```
List<Student> ls = Arrays.asList(new Student("S1", Student.Grade.A), new Student("S2", Student.Grade.A), new Student("S3", Student.Grade.C));
//INSERT CODE HERE
System.out.println(grouping);
```

Left Unanswered
You had to select 1 option

~~Map<Student.Grade, List<Student>> grouping = ls.stream().collect(~~
~~Collectors.groupingBy(Student::getGrade),~~
~~Collectors.groupingBy(Student::getName, Collectors.toList())));~~

Invalid arguments to the collect method.

Remember that Stream has only two overloaded collect methods - one that takes a Collector as an argument and another one that takes a Supplier, BiConsumer, and BiConsumer. In this option, it is trying to pass two Collectors to the collect method. Therefore, it will not compile.

1. `public <R,A> R collect(Collector<? super T,A,R> collector)`
Performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to Stream.collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

2.`public <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
Performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.

```
Map<Student.Grade, List<String>> grouping = ls.stream().collect(
        Collectors.groupingBy(Student::getGrade,
            Collectors.groupingBy(Student::getName, Collectors.toList())));
```

The right hand side of = is actually okay from a compilation perspective. It is trying to group the elements of the stream by Grade and then it is again trying to groups the elements of each grade by name. So technically, the return type of this expression would be:
`Map<Student.Grade, Map<String, List<Student>>> grouping = ...`

Even if you change the left hand side declaration as described, it will only print {C={S3=[S3:C]}, A={S1=[S1:A], S2=[S2:A]}}, which is not what is required by the question.

```
Map<Student.Grade, List<String>> grouping = ls.stream().collect(
    Collectors.groupingBy(Student::getGrade,
    Collectors.mapping(Student::getName, Collectors.toList())));
```

This code illustrates how to cascade Collectors.
Here, you are first grouping the elements by Grade and then collecting each element of a particular grade into a list after mapping it to a String. This will produce the required output.

```
Map<Student.Grade, List<String>> grouping = ls.stream().collect(
                Collectors.groupingBy(Student::getGrade,
                Collectors.mapping(Student::getName)));
```

Collectors.mapping method requires two arguments - the first argument must be a Function that maps one element type into another (here, you are mapping Student to String, which is good), and the second argument must be an appropriate Collector in which you can hold the result (here, this argument is missing). Therefore, it will not compile.

Add Note

What will the following code print?

```
Object v1 = IntStream.rangeClosed(10, 15)
        .boxed()
        .filter(x->x>12)
        .parallel()
        .findAny();

Object v2 = IntStream.rangeClosed(10, 15)
        .boxed()
        .filter(x->x>12)
        .sequential()
        .findAny();

System.out.println(v1+":"+v2);
```

(Note: < and > in the options below denote the possible output and not the sign themselves.)

You answered correctly
You had to select 1 option

- Optional[13]:Optional[13]
- <An Optional containing 13, 14, or 15>:Optional[13]
- **<An Optional containing 13, 14, or 15>:<An Optional containing 13, 14, or 15>**
- 14:13
- <13, 14, or 15>:13
- <13, 14, or 15>:<13, 14, or 15>
- An exception at run time.

Explanation

Since the first stream is made parallel, it may be partitioned into multiple pieces and each piece may be processed by a different thread. findAny may, therefore, return a value from any of those partitions. Hence, any number from 13 to 15 may be printed.

The second stream is sequential and therefore, ideally, findAny should return the first element. However, findAny is deliberately designed to be non-deterministic. Its API specifically says that it may return any element from the stream. If you want to select the first element, you should use findFirst.

Further findAny returns Optional object. Therefore, the output will be Optional[13] instead of just 13 (or any other number).

Add Note

Given:

```
interface Account{
  public default String getId(){
     return "0000";
  }
}

interface PremiumAccount extends Account{
   //INSERT CODE HERE
}
```

Which of the following options can be inserted in PremiumAccount independent of each other?

You answered incorrectly

You had to select 2 options

- [ ]
```
static String getId(){
  return "1111";
}
```

In case of classes, you cannot override a static method with a non-static method and vice-versa.
However, in case of interfaces, it is possible for a sub interface to have a non-static (i.e. default) method with the same signature as that of a static method of a super interface. This is because static methods are not inherited in any sense in the sub-interface and that is why it is allowed to declare a default method with the same signature. The reverse is not true though. That is, you cannot have a static method in a sub-interface with the same signature as that of a default method in a super interface.

- [x] `String getId();`

An interface can redeclare a default method and also make it abstract.

- [ ] ~~`default String getId(){ return "1111"; }`~~

An interface can redeclare a default method and provide a different implementation.

- [ ] ~~`abstract static String getName();`~~

1. static methods can never be abstract (neither in an interface not in a class).
2. An interface can have a static method but the method must have a body.

- [x] `static String getName();`

An interface can have a static method but the method must have a body.

- [ ] ~~`default String getName();`~~

A default method must have a body.

Add Note

Which of these statements about interfaces are true?

You answered incorrectly
You had to select 3 options

- [ ] Interfaces are always abstract.
- [x] An interface can have static methods.

Java 8 allows interfaces to have static methods as well as default methods.

- [x] All methods in an interface are abstract although you need not declare them to be so.

An interface may have default methods. A method marked default is considered a non-abstract instance method. A non-abstract class that implements this interface doesn't necessarily have to implement a default method.

- [ ] Fields of an interface may be declared as transient or volatile but not synchronized.

All fields of an interface are public, static, and final. Therefore, volatile, transient, and synchronized do not make sense for such fields.

- [ ] Interfaces cannot be final.
- [x] In Java 8, interfaces allow multiple implementation inheritance through default methods.

They don't. You cannot have a class that implements two interfaces where both the interfaces contain a default method with the same signature unless the class provides an implementation for that method itself. For example, in the following code, class C will not compile:

```
interface I1{
  public default void m1(){
     System.out.println("in I1.m1");
  }
}
interface I2{
  public default void m1(){
     System.out.println("in I2.m1");
  }
}

class CI implements I1, I2{ //This class will not compile.
}
class C2 implements I1, I2{ //This class will compile because it provides its own implementation of m1.
  public void m1(){
     System.out.println("in C2.m1");
  }
}
```

You can have a class inherit a method with the same signature from an interface and a superclass though. This is allowed because the superclass's version always overrides the interface's version. The class doesn't get two implementations. It gets only the version from super class.

Add Note

Given that java.lang.String has two overloaded toUpperCase methods - toUpperCase() and toUpperCase(Locale), consider the following code:

```
String name = "bob";
String val = null;
//Insert code here
System.out.print(val);
```

Which of the following code fragments can be inserted in the above code so that it will print BOB?

You answered incorrectly
You had to select 2 options

- [x] `Supplier<String> s = name::toUpperCase;`
  `val = s.get();`
- [ ] `Supplier<String> s = name::toUpperCase;`
  `val = s.apply();`

The name of the method in Supplier is get (not apply).

- [ ] `Function<String> f = name::toUpperCase;`
  `val = f.get();`

Function takes one argument and returns a value. So `Function<Type>` will not compile. It should actually be Function<T, R>. For example, Function<Locale, String>. Where Locale is the type of the input and String is the type of the return value.

If you want to make use of toUpperCase(Locale) method, you should do:
`Function<Locale, String> f1 = name::toUpperCase;`
You can then use it like this:
`val = f1.apply(Locale.UK);`

- [x] `Function<String> f = name::toUpperCase;`
  `val = f.apply();`
- [ ] `Function<String, Locale> f= name::toUpperCase;`
  `val = f.apply();`

1. It should be `Function<Locale, String> f =` instead of `Function<String, Locale> f =` because you want this function to return a String (not a Locale). The type of the return value is specified at the last position.
2. Function expects an argument to be passed. Thus, it should be `val = f.apply(Locale.US);` instead of just `val = f.apply;`.

- [ ] Only one of the above is correct.

Add Note

Q 33 of 60 QID : enthuware.ocpjp.v8.2.1876

Hide Section/Toughness Mark

Which of the following standard functional interfaces is most suitable to process a large collection of int primitives and return processed data for each of them?

Left Unanswered
You had to select 1 option

- Function<Integer>
- IntFunction
- Consumer<Integer>
- IntConsumer
- Predicate<Integer>

Explanation

Using the regular functional interfaces by parameterizing them to Integer is inefficient as compared to using specially designed interfaces for primitives because they avoid the cost of boxing and unboxing the primitives.
Now, since the problem statement requires something to be returned after processing each int, you need to use a Function instead of a Consumer or a Predicate.

Therefore, IntFunction is most appropriate in this case.

Add Note

Assuming that the local time when the following code is executed is 9.30 AM, what will it print?

```
LocalTime now = LocalTime.now();
LocalTime gameStart = LocalTime.of(10, 15);
long timeConsumed = 0;
long timeToStart = 0;
if(now.isAfter(gameStart)){
    timeConsumed = gameStart.until(now, ChronoUnit.HOURS);
}else{
    timeToStart = now.until(gameStart, ChronoUnit.HOURS);
}
System.out.println(timeToStart + " " + timeConsumed);
```

You answered correctly

You had to select 1 option

- 0 45
- **0 0**
- 45 0
- None of the above

Explanation

1. The `isAfter` method of LocalTime returns true only if this LocalTime is after the passed LocalTime. (I.e. if both are same, then it will return false)
In this case, it is not. Therefore, `timeConsumed` will remain 0.
2. The `until` method return the difference between the two time periods in given units. Here, the difference is 45 minutes but the unit is HOURS, therefore, it will return 0.

The API description of until method explains how it works:

```
public long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time until another time in terms of the specified unit.
This calculates the amount of time between two LocalTime objects in terms of a single TemporalUnit. The start and end points are this and the specified time. The result will be negative if the end is before the start. The Temporal passed to this method is converted to a LocalTime using LocalTime.from(TemporalAccessor). For example, the amount in hours between two times can be calculated using startTime.until(endTime, HOURS).
The calculation returns a whole number, representing the number of complete units between the two times. For example, the amount in hours between 11:30 and 13:29 will only be one hour as it is one minute short of two hours.
There are two equivalent ways of using this method. The first is to invoke this method. The second is to use TemporalUnit.between(Temporal, Temporal):

```
// these two lines are equivalent
amount = start.until(end, MINUTES);
amount = MINUTES.between(start, end);
```

The choice should be made based on which makes the code more readable.

Add Note

Previous Next Review Discuss

Given the following code:

```
public String getDateString(LocalDateTime ldt){
   return DateTimeFormatter.ISO_ZONED_DATE_TIME.format(ldt);
}
```

Which of the following statements are correct?

You answered correctly
You had to select 1 option

- (•) The code will compile but will always throw a `DateTimeException` (or its subclass) at run time.

Note that LocalDateTime class does not contain Zone information but ISO_ZONED_DATE_TIME requires it. Thus, it will throw the following exception:

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: OffsetSeconds
```

UnsupportedTemporalTypeException extends DateTimeException.

- ( ) DateTimeException must either be caught or declared in the throws clause of this method.

`DateTimeException` extends `RuntimeException`, so it need not be caught or declared in the throws clause.

- ( ) The method parameter type must be changed from `LocalDateTime` to `ZonedDateTime` for it to compile.

Although it is true that this code will never work at runtime, it will compile fine as it is.

- ( ) It will return the date string as per the default time zone of the system on which it is run.

Add Note

Given that daylight Savings Time starts on March 8th at 2 AM in US/Eastern time zone. (As a result, 2 AM becomes 3 AM.), what will the following code print?

```
LocalDateTime ld1 = LocalDateTime.of(2015, Month.MARCH, 8, 2, 0);
ZonedDateTime zd1 = ZonedDateTime.of(ld1, ZoneId.of("US/Eastern"));
LocalDateTime ld2 = LocalDateTime.of(2015, Month.MARCH, 8, 3, 0);
ZonedDateTime zd2 = ZonedDateTime.of(ld2, ZoneId.of("US/Eastern"));
long x = ChronoUnit.HOURS.between(zd1, zd2);
System.out.println(x);
```

Left Unanswered

You had to select 1 option

- 1
- -1
- 0
- 2
- -2
- It will not compile.

Explanation

Think of it as follows -
The time difference between two dates is simply the amount of time you need to go from date 1 to date 2.
So if you want to go from 2AM to 3AM, how many hours do you need? On a regular day, you need 1 hour. That is, if you add 1 hour to 2AM, you will get 3AM. However, as given in the problem statement, at the time of DST change, 2 AM becomes 3AM. That means, even though your local date time is 2 AM, your ZonedDateTime is actually 3AM. Therefore, you are already at 3AM, which means, there is no time difference between 2 AM and 3 AM. The answer is, therefore, 0.

Add Note

Q 60 of 60 QID : enthuware.ocpjp.v8.2.1159

Hide Section/Toughness Mark

Identify valid statements.

You answered incorrectly
You had to select 3 options

- [x] `Locale myLocale = System.getDefaultLocale();`

There is no such method in System class.

- [ ] `Locale myLocale = Locale.getDefaultLocale();`
- [ ] `Locale myLocale = Locale.getDefault();`
- [x] `Locale myLocale = Locale.US;`

Locale class has several static constants for standard country locales.

- [ ] `Locale myLocale = Locale.getInstance();`

There is no getInstance() method in Locale.

- [x] `Locale myLocale = new Locale("ru", "RU");`

You don't have to worry about the actual values of the language and country codes. Just remember that both are two lettered codes and country codes are always upper case.

Add Note

Q 60 of 60 QID : enthuware.ocpjp.v8.2.1159

Hide Section/Toughness Mark

Identify valid statements.

You answered incorrectly
You had to select 3 options

- [x] `Locale myLocale = System.getDefaultLocale();`

There is no such method in System class.

- [ ] `Locale myLocale = Locale.getDefaultLocale();`
- [x] `Locale myLocale = Locale.getDefault();`
- [ ] `Locale myLocale = Locale.US;`

Locale class has several static constants for standard country locales.

- [ ] `Locale myLocale = Locale.getInstance();`

There is no getInstance() method in Locale.

- [x] `Locale myLocale = new Locale("ru", "RU");`

You don't have to worry about the actual values of the language and country codes. Just remember that both are two lettered codes and country codes are always upper case.

Add Note

What will the following code print?

```
Map<String, Integer> map1 = new HashMap<>();
map1.put("a", 1);
map1.put("b", 1);
map1.merge("b", 1, (i1, i2)->i1+i2);
map1.merge("c", 3, (i1, i2)->i1+i2);
System.out.println(map1);
```

Left Unanswered

You had to select 1 option

- {a=1, b=2, c=3}
- {a=1, b=1, c=3}
- {a=1, b=2}
- A NullPointerException will be thrown at run time.

Explanation

The JavaDoc API description explains exactly how the merge method works. You should go through it as it is important for the exam.

```
public V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
```

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a String msg to a value mapping:
map.merge(key, msg, String::concat)

If the function returns null the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Parameters:
key - key with which the resulting value is to be associated value - the non-null value to be merged with the existing value associated with the key or, if no existing value or a null value is associated with the key, to be associated with the key remappingFunction - the function to recompute a value if present

Returns:
the new value associated with the specified key, or null if no value is associated with the key

Throws:
UnsupportedOperationException - if the put operation is not supported by this map (optional)
ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)
NullPointerException - if the specified key is null and this map does not support null keys or the value or remappingFunction is null

Add Note

How will you initialize a SimpleDateFormat object so that the following code will print the number of the month (i.e. 02 for Feb, 12 for Dec, and so on) and a two digit calendar year of any given date?

```
System.out.println(sdf.format(new Date()));
```

You answered correctly
You had to select 1 option

- `SimpleDateFormat sdf = new SimpleDateFormat("mm/yy", Locale.US);`

lower case m is for minutes. So mm will print the current minute and not the current month.

- (selected) `SimpleDateFormat sdf = new SimpleDateFormat("MM/yy", Locale.US);`

Upper case M is for Month and lower case y is for Calendar Year.

- `SimpleDateFormat sdf = new SimpleDateFormat("mm/YY", Locale.US);`

Upper case Y is for a week year. The first and last days of a week year may have different calendar year values and so if you use upper case Y, it may not print the correct calendar year depending on the date.

For the purpose of the exam, just remember that you need to use lower case y for year.

- `SimpleDateFormat sdf = new SimpleDateFormat("MM/YY", Locale.US);`

Explanation

For the purpose of the exam, you need to know the basic codes for printing out a date. The important ones are m, M, d, D, y, s, S, h, H, and z. The following table shows all the codes (Please check http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html for details) :

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

Pattern letters are usually repeated, as their number determines the exact presentation.

Examples

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"	2001-07-04T12:08:56.235-07:00
"YYYY-'W'ww-u"	2001-W27-3

Given:

```
List<Integer> ls = Arrays.asList(3,4,6,9,2,5,7);
System.out.println(ls.stream().reduce(Integer.MIN_VALUE, (a, b)->a>b?a:b)); //1
System.out.println(ls.stream().max(Integer::max).get()); //2
System.out.println(ls.stream().max(Integer::compare).get()); //3
System.out.println(ls.stream().max((a, b)->a>b?a:b)); //4
```

Which of the above statements will print 9?

You answered incorrectly
You had to select 1 option

- 1 and 4
- 2 and 3
- 1 and 3
- 2, 3, and 4
- All of them.
- None of them.

Explanation

The code will print:

```
9
3
9
Optional[3]
```

You need to understand the following points to answer this question:

1. The reduce method needs a `BinaryOperator`. This interface is meant to consume two arguments and produce one output. It is applied repeatedly on the elements in the stream until only one element is left. The first argument is used to provide an initial value to start the process. (If you don't pass this argument, a different reduce method will be invoked and that returns an Optional object.)

2. The `Stream.max` method requires a `Comparator`. All you need to implement this interface using a lambda expression is a reference to any method that takes two arguments and returns an int. The name of the method doesn't matter. That is why it is possible to pass the reference of `Integer's max` method as an argument to `Stream's max` method. However, `Integer.max` works very differently from `Integer.compare`. The `max` method returns the maximum of two numbers while the `compare` method returns a difference between two numbers. Therefore, when you pass `Integer::max` to `Stream's max`, you will not get the correct maximum element from the stream. That is why //2 will compile but will not work correctly.

//4 is basically same as //2. It will not work correctly for the same reason.

Add Note

Given:

```
interface Carnivore{
    default int calories(List<String> food){
        return food.size()*100;
    }
    int eat(List<String> foods);
}
class Tiger implements Carnivore{
    public int eat(List<String> foods){
        System.out.println("Eating "+foods);
        return foods.size()*200;
    }
}
public class TestClass {
    public static int size(List<String> names){
        return names.size()*2;
    }
    public static void process(List<String> names, Carnivore c){
        c.eat(names);
    }

    public static void main(String[] args) {
        List<String> fnames = Arrays.asList("a", "b", "c");
        Tiger t = new Tiger();

        INSERT CODE HERE

   }
}
```

Which of the following options can be inserted independent of each other in the code above without any compilation error?

Left Unanswered

You had to select 3 options

- [ ] `process(fnames, t::eat);`
- [ ] `process(fnames, t::calories);`
- [ ] `process(fnames, TestClass::size);`
- [ ] `process(fnames, Carnivore::calories);`
- [ ] `process(fnames, Tiger::eat);`

Explanation

Don't be confused by twisted code. `process(List<String> names, Carnivore c)` expects a `List<String>` and a `Carnivore` instance as arguments. `Carnivore` has exactly one abstract method and therefore it is a functional interface. You can either pass a `Carnivore` instance explicitly or pass a reference to a method that matches the parameter list of `Carnivore`'s abstract method `eat(List<String> foods);`.

`t::eat`, `t::calories`, and `TestClass::size` are all valid method references with the exact same parameter requirements and are therefore valid.

`Carnivore::calories` is invalid because `Carnivore` is an interface. It does not refer to any object upon which `calories` method can be invoked. `t::calories`, on the other hand is valid because t does refer to an object upon which `calories` method can be invoked. `t::eat` is valid for the same reason.

`Tiger::eat` is a valid method reference that can mean to refer either to a static method eat of Tiger class or to an instance method of any arbitraty instance of Tiger class. Which meaning is implied depends on the context in which it is used. Here, the context does not supply any instance of Tiger class. Therefore, `Tiger::eat` will refer to a static method eat. But there is no such static method in `Tiger` class. Therefore, it is invalid in this context. To use `Tiger::eat`, you need a reference to a Tiger instance, which is not available here. `TestClass::size`, on the other hand, is a static method of `TestClass`, which means you don't need an object of `TestClass` to invoke this method and that is why it is valid.

Add Note

Consider the following code:

```
public class FileCopier {
    public static void copy(String records1, String records2)  {
        try {
            InputStream is = new FileInputStream(records1);
            OutputStream os = new FileOutputStream(records2);
            byte[] buffer = new byte[1024];
            int bytesRead = 0;
            while ((bytesRead = is.read(buffer)) != -1) {
                os.write(buffer, 0, bytesRead);
                System.out.println("Read and written bytes " + bytesRead);
            }
        } catch (FileNotFoundException | IndexOutOfBoundsException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        copy("c:\\temp\\test1.txt", "c:\\temp\\test2.txt");
    }
}
```

Assuming appropriate import statements and the existence of both the files, what will happen when the program is compiled and run?

You answered incorrectly

You had to select 1 option

- The program will not compile because it does not handle exceptions correctly.

Remember that most of the I/O operations (such as opening a stream on a file, reading or writing from/to a file) throw `IOException` and this code does not handle `IOException`. `FileNotFoundException` is a subclass of `IOException` and `IndexOutOfBoundsException` is subclass of `RuntimeException`.
The code can be fixed by replacing both the exceptions with `IOException`.

- The program will compile but will throw an exception at run time.
- The program will not compile because the catch clause is used incorrectly.

The catch clause used here is valid. FYI, the exception parameter in a multi-catch clause is implicitly final. Thus, unlike in a regular catch block, it cannot be reassigned.

- It will compile and run without any error or exception.

Add Note

Consider the following method code:

```
public static void copy(String records1, String records2) {
        try (
                InputStream is = new FileInputStream(records1);
                OutputStream os = new FileOutputStream(records2);  ) {

            byte[] buffer = new byte[1024];
            int bytesRead = 0;
            while ((bytesRead = is.read(buffer)) != -1) {
                os.write(buffer, 0, bytesRead);
                System.out.println("Read and written bytes " + bytesRead);
            }
        }
        catch (  *INSERT CODE HERE*  e ) { //LINE 100
        }

    }
```

What can be inserted at //LINE 100 to make the method compile?

You answered incorrectly
You had to select 1 option

(•) `Exception|IOException`

`IOException` is a subclass of `Exception`. You cannot include classes that are related by inheritance in the same multi-catch block.

( ) `FileNotFoundException|SecurityException|IllegalArgumentException`

Note that most commonly used methods in Java API that deal with reading or writing files have `java.io.IOException` in their throws clause. So you must handle this exception. At run time, more specific exceptions such as `FileNotFoundException` are actually thrown depending on the actual cause of the problem.

( ) `FileNotFoundException|IOException`

`FileNotFountException` is a subclass of `IOException`. You cannot include classes that are related by inheritance in the same multi-catch block.

( ) `IOException|RuntimeException`

( ) `IOException|NoSuchFileException`

Remember that `java.io.IOException` is a superclass of all exceptions under `java.nio.file` package. This combination, therefore, is invalid.

Add Note

What will the following code fragment print?

```
Path p1 = Paths.get("photos\\..\\beaches\\.\\calangute\\a.txt");
Path p2 = p1.normalize();
Path p3 = p1.relativize(p2);
Path p4 = p2.relativize(p1);

System.out.println(
    p1.getNameCount()+" "+p2.getNameCount()+" "+
    p3.getNameCount()+" "+p4.getNameCount());
```

You answered incorrectly
You had to select 1 option

- (•) 6 4 10 10
- ( ) ~~7 4 11 10~~
- ( ) ~~7 3 8 9~~
- ( ) 6 3 9 9

1. p1 has 6 components and so `p1.getNameCount()` will return 6.
2. `normalize` applies all the .. and . contained in the path to the path. Therefore, p2 contains `beaches\calangute\a.txt`, that is 3 components.
3. p3 contains `..\..\..\..\..\..\beaches\calangute\a.txt`, that is 9 components.
4. p4 contains `..\..\..\photos\..\beaches\.\calangute\a.txt`, that is 9 components.

Explanation

You need to understand how relativize works for the purpose of the exam. The basic idea of relativize is to determine a path, which, when applied to the original path will give you the path that was passed. For example, "a/b" relativize "c/d" is "../../c/d" because if you are in directory b, you have to go two steps back and then one step forward to c and another step forward to d to be in d. However, "a/c" relativize "a/b" is "../b" because you have to go only one step back to a and then one step forward to b.

Please go through the following description of relativize() method, which explains how it works in more detail.

public Path relativize(Path other)
Constructs a relative path between this path and a given path. Relativization is the inverse of resolution. This method attempts to construct a relative path that when resolved against this path, yields a path that locates the same file as the given path. For example, on UNIX, if this path is "/a/b" and the given path is "/a/b/c/d" then the resulting relative path would be "c/d".

Where this path and the given path do not have a root component, then a relative path can be constructed.

A relative path cannot be constructed if only one of the paths have a root component.

Where both paths have a root component then it is implementation dependent if a relative path can be constructed.

If this path and the given path are equal then an empty path is returned.

For any two normalized paths p and q, where q does not have a root component,
p.relativize(p.resolve(q)).equals(q)

When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the same file as other is implementation dependent. For example, if this path is "/a/b" and the given path is "/a/x" then the resulting relative path may be "../x". If "b" is a symbolic link then is implementation dependent if "a/b/../x" would locate the same file as "/a/x".

Add Note

Consider the following code:

```
import java.util.*;
import java.text.*;

public class TestClass
{
    public static void main(String[] args) throws Exception
    {
        double amount = 53000.35;

        Locale jp = new Locale("jp", "JP");

        //1 create formatter here.

        System.out.println(   formatter.format(amount)  );
    }
}
```

How will you create formatter using a factory at //1 so that the output is in Japanese Currency format?

You answered incorrectly
You had to select 2 options

☐ `NumberFormat formatter =  NumberFormat.getCurrencyFormatter(jp);`

☐ ~~`NumberFormat formatter =  new DecimalFormat(jp);`~~

1. `DecimalFormat` has no constructor that takes a `Locale`.
2. Creating an object using new as done in this option means you are not using a factory. Remember, using a factory to get an object, usually means calling `getInstance()` or `getXXXInstance()` method on a Factory class such as `NumberFormat` or `DateFormat`.

☐ `Format formatter =  NumberFormat.getCurrencyInstance(jp);`

This is valid because `java.text.NumberFormat` extends from `java.text.Format`. The return type of the method `getCurrencyInstance()` is `NumberFormat`.

☑ `NumberFormat formatter =  DecimalFormat.getCurrencyInstance(jp);`

`getCurrencyInstance` is actually defined in `NumberFormat`. However, since `DecimalFormat` extends `NumberFormat`, this is valid.

To format a number in currency format, you should use `getCurrencyInstance()` instead of `getInstance()` or `getNumberInstance()`.

This will print : `JPY 53,000`

☑ `NumberFormat formatter =  NumberFormat.getInstance(jp);`

`getInstance(Locale )` is a valid factory method in `NumberFormat` class but it will not not format the given number as per the currency.

☐ `NumberFormat formatter =  new DecimalFormat("#.00");`

While it is a valid way to create a DecimalFormat object, it is not valid for two reasons:
1. We need a currency formatter and not just a simple numeric formatter.
2. This is not using a factory to create the formatter object.

Explanation

To obtain a `NumberFormat` for a specific locale, including the default locale, call one of `NumberFormat`'s factory methods, such as `getInstance()`. In general, do not call the `DecimalFormat` constructors directly, since the `NumberFormat` factory methods may return subclasses other than `DecimalFormat`. If you need to customize the format object, do something like this:

```
NumberFormat f = NumberFormat.getInstance(loc);
if (f instanceof DecimalFormat) {
     ((DecimalFormat) f).setDecimalSeparatorAlwaysShown(true);
}
```

Add Note

You want to create a new file. If the file already exists, you want the new file to overwrite the existing one (the content of the existing file, if any, should go away). Which of the following code fragments will accomplish this?

You answered incorrectly
You had to select 1 option

```
Path myfile = Paths.get("c:\\temp\\test.txt");
BufferedWriter br = Files.newBufferedWriter(myfile, Charset.forName("UTF-8"),
    new OpenOption[] {StandardOpenOption.CREATE});
```

CREATE option will cause the file to be overwritten from the start of the file and it will NOT get rid of the existing content.

```
Path myfile = Paths.get("c:\\temp\\test.txt");
BufferedWriter br = Files.newBufferedWriter(myfile, Charset.forName("UTF-8"),
    new OpenOption[] {StandardOpenOption.CREATE_NEW});
```

CREATE_NEW will cause an exception to be thrown if the file already exists.

```
Path myfile = Paths.get("c:\\temp\\test.txt");
BufferedWriter br = Files.newBufferedWriter(myfile, Charset.forName("UTF-8"),
    new OpenOption[] {StandardOpenOption.WRITE});
```

WRITE will cause an exception to be thrown if the file does not exist. It will not get rid of the existing content either.

```
Path myfile = Paths.get("c:\\temp\\test.txt");
BufferedWriter br = Files.newBufferedWriter(myfile, Charset.forName("UTF-8"),
    new OpenOption[] {StandardOpenOption.TRUNCATE_EXISTING, StandardOpenOption.CREATE_NEW});
```

While TRUNCATE_EXISTING means that all existing content will be removed from the file, CREATE_NEW implies that the file should not exist already. Thus, if the file already exists, it will throw `java.nio.file.FileAlreadyExistsException`.

```
Path myfile = Paths.get("c:\\temp\\test.txt");
BufferedWriter br = Files.newBufferedWriter(myfile, Charset.forName("UTF-8"),
    new OpenOption[] {StandardOpenOption.TRUNCATE_EXISTING, StandardOpenOption.CREATE});
```

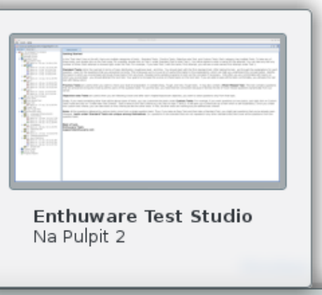
If the file already exists, TRUNCATE_EXISTING will take care of the existing content. If the file does not exist, CREATE will ensure that it is created.

Explanation

`StandardOpenOption` gives you several ways of opening a file. You can mix and match the options to achieve the desired objective. We encourage you to run the following code with different open options and observe how it behaves. Observe that some combinations such as `StandardOpenOption.CREATE, StandardOpenOption.READ` do not make sense if put together and therefore an `IllegalArgumentException` will be thrown in such cases. There are questions on the exam that expect you to know such combinations.

```
import java.io.BufferedWriter;
import java.nio.charset.Charset;
import java.nio.file.*;
public class TestClass
{

   public static void main(String args[]) throws Exception
   {
       Path writeFile = Paths.get("c:\\temp\\test.txt");
       BufferedWriter br = Files.newBufferedWriter(writeFile,
           Charset.forName("UTF-8"),
           new OpenOption[] {StandardOpenOption.CREATE, StandardOpenOption.READ});
        br.write("This text file is created using Path API");
        br.flush();
        br.close();
   }
}
```

Add Note



Next Review Discuss*

Which of the following statements are valid usages of StandardOpenOption options that control how the file is opened?

Left Unanswered
You had to select 4 options

- [ ] `new OpenOption[]{StandardOpenOption.WRITE, StandardOpenOption.DELETE_ON_CLOSE}`

- [ ] `new OpenOption[]{StandardOpenOption.READ, StandardOpenOption.DELETE_ON_CLOSE}`

This is a valid combination on Java 8.

- [ ] `new OpenOption[]{StandardOpenOption.DELETE_ON_CLOSE, StandardOpenOption.TRUNCATE_EXISTING}`

This is a valid combination but will throw java.nio.file.NoSuchFileException if the file does not exist.

- [ ] `new OpenOption[]{StandardOpenOption.APPEND, StandardOpenOption.CREATE_NEW}`

This throws java.lang.UnsupportedOperationException: 'APPEND' not allowed.

- [ ] `new OpenOption[]{StandardOpenOption.READ, StandardOpenOption.SYNC}`

Ideally, this should be an invalid combination (because when a file is opened for READ, there is nothing to synch) but it works.

Explanation

Observe that some combinations such as `CREATE` and `READ` do not make sense if put together and therefore an `IllegalArgumentException` will be thrown in such cases. There are questions on the exam that expect you to know such combinations.

JavaDoc API does not explicitly mention which combinations are invalid and so in many cases it is not easy to spot a valid/invalid combination. For example, if you want to truncate a file, then you must open it with an option that allows writing. Thus, `READ` and `TRUNCATE_EXISTING` (or `WRITE, APPEND, or DELETE_ON_CLOSE`) should not go together. However, on Java 8, `READ` and `DELETE_ON_CLOSE` work fine together.

Please note that the validity of these combinations also depends on the OS and the file system used. So it is not possible to precisely determine which combinations will always be invalid.

The following is a list of constants provided by `java.nio.file.StandardOpenOption`:

`APPEND`
If the file is opened for `WRITE` access then bytes will be written to the end of the file rather than the beginning.

`CREATE`
Create a new file if it does not exist.

`CREATE_NEW`
Create a new file, failing if the file already exists.

`DELETE_ON_CLOSE`
Delete on close.

`DSYNC`
Requires that every update to the file's content be written synchronously to the underlying storage device.

`READ`
Open for read access.

`SPARSE`
Sparse file.

`SYNC`
Requires that every update to the file's content or metadata be written synchronously to the underlying storage device.

`TRUNCATE_EXISTING`
If the file already exists and it is opened for `WRITE` access, then its length is truncated to 0.

`WRITE`
Open for write access.

Add Note

Which of the following statements are valid usages of `StandardOpenOption` options that determine how the file is opened?

You answered incorrectly
You had to select 3 options

- [x] `new OpenOption[]{StandardOpenOption.WRITE, StandardOpenOption.DSYNC}`
- [ ] `new OpenOption[]{StandardOpenOption.READ, StandardOpenOption.APPEND}`

This is an invalid combination.

- [ ] `new OpenOption[]{StandardOpenOption.APPEND, StandardOpenOption.TRUNCATE_EXISTING}`

Ideally, this should be an invalid combination (because when a file is opened for READ, there is nothing to synch) but it works.

- [x] `new OpenOption[]{StandardOpenOption.APPEND, StandardOpenOption.SYNC}`
- [ ] `new OpenOption[]{StandardOpenOption.READ, StandardOpenOption.SYNC}`

This is an invalid combination.

Explanation

Observe that some combinations such as `CREATE` and `READ` do not make sense if put together and therefore an `IllegalArgumentException` will be thrown in such cases. There are questions on the exam that expect you to know such combinations.

JavaDoc API does not explicitly mention which combinations are invalid but it many cases it is easy to spot an invalid combination. For example, if you want to truncate a file, then you must open it with an option that allows writing. Thus, `READ` and `TRUNCATE_EXISTING` (or `WRITE, APPEND,` or `DELETE_ON_CLOSE`) cannot go together. `READ` and `SYNC` (or `DSYNC`) cannot go together either because when a file is opened for `READ`, there is nothing to synch.

Please note that the validity of these combinations also depends on the OS and the file system used. So it is not possible to precisely determine which combinations will always be invalid.

The following is a list of constants provided by `java.nio.file.StandardOpenOption`:

`APPEND`
If the file is opened for `WRITE` access then bytes will be written to the end of the file rather than the beginning.

`CREATE`
Create a new file if it does not exist.

`CREATE_NEW`
Create a new file, failing if the file already exists.

`DELETE_ON_CLOSE`
Delete on close.

`DSYNC`
Requires that every update to the file's content be written synchronously to the underlying storage device.

`READ`
Open for read access.

`SPARSE`
Sparse file.

`SYNC`
Requires that every update to the file's content or metadata be written synchronously to the underlying storage device.

`TRUNCATE_EXISTING`
If the file already exists and it is opened for `WRITE` access, then its length is truncated to 0.

`WRITE`
Open for write access.

Square brackets [] are used to specify multiple options for a single character. Therefore, the above pattern will match a.h, a.t, a.m, a.,, a.x and a.l but not a.htm or a.xml

```
PathMatcher pm = FileSystems.getDefault().getPathMatcher("glob:**.[htm*,xml]");
```

```
PathMatcher pm = FileSystems.getDefault().getPathMatcher("glob:**.htm,html,xml");
```

This will match a path ending with .htm,html,xml

Explanation

Please go through the following JavaDoc API description for FileSystem.getPathMatcher method. This is important for the exam.

public abstract PathMatcher getPathMatcher(String syntaxAndPattern)

Returns a PathMatcher that performs match operations on the String representation of Path objects by interpreting a given pattern. The syntaxAndPattern parameter identifies the syntax and the pattern and takes the form:
syntax:pattern

where ':' stands for itself.
A FileSystem implementation supports the "glob" and "regex" syntaxes, and may support others. The value of the syntax component is compared without regard to case.
When the syntax is "glob" then the String representation of the path is matched using a limited pattern language that resembles regular expressions but with a simpler syntax. For example:

*.java
Matches a path that represents a file name ending in .java

.
Matches file names containing a dot

*.{java,class}
Matches file names ending with .java or .class

foo.?
Matches file names starting with foo. and a single character extension

/home/*/*
Matches /home/gus/data on UNIX platforms

/home/**
Matches /home/gus and /home/gus/data on UNIX platforms

C:*
Matches C:\foo and C:\bar on the Windows platform (note that the backslash is escaped; as a string literal in the Java Language the pattern would be "C:*")

The following rules are used to interpret glob patterns:

The * character matches zero or more characters of a name component without crossing directory boundaries.
The ** characters matches zero or more characters crossing directory boundaries.
The ? character matches exactly one character of a name component.
The backslash character (\) is used to escape characters that would otherwise be interpreted as special characters. The expression \\ matches a single backslash and "\{" matches a left brace for example.

The [ ] characters are a bracket expression that match a single character of a name component out of a set of characters. For example, [abc] matches "a", "b", or "c". The hyphen (-) may be used to specify a range so [a-z] specifies a range that matches from "a" to "z" (inclusive). These forms can be mixed so [abce-g] matches "a", "b", "c", "e", "f" or "g". If the character after the [is a ! then it is used for negation so [!a-c] matches any character except "a", "b", or "c".
Within a bracket expression the *, ? and \ characters match themselves. The (-) character matches itself if it is the first character within the brackets, or the first character after the ! if negating.

The { } characters are a group of subpatterns, where the group matches if any subpattern in the group matches. The "," character is used to separate the subpatterns. Groups cannot be nested.

Leading period/dot characters in file name are treated as regular characters in match operations. For example, the "*" glob pattern matches file name ".login". The Files.isHidden method may be used to test whether a file is considered hidden.

All other characters match themselves in an implementation dependent manner. This includes characters representing any name-separators.

The matching of root components is highly implementation-dependent and is not specified.

When the syntax is "regex" then the pattern component is a regular expression as defined by the Pattern class.

For both the glob and regex syntaxes, the matching details, such as whether the matching is case sensitive, are implementation-dependent and therefore not specified.

Parameters:
syntaxAndPattern - The syntax and pattern

Returns:
A path matcher that may be used to match paths against the pattern

Throws:
IllegalArgumentException - If the parameter does not take the form: syntax:pattern PatternSyntaxException - If the pattern is invalid UnsupportedOperationException - If the pattern syntax is not known to the implementation

Which of the following is true about the PathMatcher returned by `FileSystem.getPathMatcher()`?

You answered correctly

You had to select 1 option

- It provides methods that relieve a developer from writing expressions for matching paths.

A developer still needs to provide a regular expression pattern or a glob pattern that drives the PathMatcher.

- (selected) It can match Paths using a regular expression or a glob pattern depending on how it is acquired.

You can get a PathMatcher using either of the expressions. For example,

```
FileSystems.getDefault().getPathMatcher("glob:*.java"); //Uses a glob pattern
FileSystems.getDefault().getPathMatcher("regex:a.*b\.txt"); //uses a regular expression pattern
```

- It provides overloaded matches methods to suite different requirements.

It has only one matches method: `boolean matches(Path path)`

- It cannot be used for paths from multiple file systems.

A PathMatcher doesn't care about file systems. It just takes the Path objects and checks whether it matches the expression. It doesn't actually access the file or the file system.

Add Note

Square brackets [] are used to specify multiple options for a single character. Therefore, the above pattern will match a.h, a.t, a.m, a.,, a.x and a.l but not a.htm or a.xml

- (●) `PathMatcher pm = FileSystems.getDefault().getPathMatcher("glob:*.[htm*,xml]");`
- ( ) ~~`PathMatcher pm = FileSystems.getDefault().getPathMatcher("glob:*.htm,html,xml");`~~

This will match a path ending with .htm,html,xml

Explanation

Please go through the following JavaDoc API description for FileSystem.getPathMatcher method. This is important for the exam.

public abstract PathMatcher getPathMatcher(String syntaxAndPattern)

Returns a PathMatcher that performs match operations on the String representation of Path objects by interpreting a given pattern. The syntaxAndPattern parameter identifies the syntax and the pattern and takes the form:
syntax:pattern

where ':' stands for itself.
A FileSystem implementation supports the "glob" and "regex" syntaxes, and may support others. The value of the syntax component is compared without regard to case.
When the syntax is "glob" then the String representation of the path is matched using a limited pattern language that resembles regular expressions but with a simpler syntax. For example:

*.java
Matches a path that represents a file name ending in .java

.
Matches file names containing a dot

*.{java,class}
Matches file names ending with .java or .class

foo.?
Matches file names starting with foo. and a single character extension

/home/*/*
Matches /home/gus/data on UNIX platforms

/home/**
Matches /home/gus and /home/gus/data on UNIX platforms

C:*
Matches C:\foo and C:\bar on the Windows platform (note that the backslash is escaped; as a string literal in the Java Language the pattern would be "C:*")

The following rules are used to interpret glob patterns:

The * character matches zero or more characters of a name component without crossing directory boundaries.
The ** characters matches zero or more characters crossing directory boundaries.
The ? character matches exactly one character of a name component.
The backslash character (\) is used to escape characters that would otherwise be interpreted as special characters. The expression \\ matches a single backslash and "\{" matches a left brace for example.

The [ ] characters are a bracket expression that match a single character of a name component out of a set of characters. For example, [abc] matches "a", "b", or "c". The hyphen (-) may be used to specify a range so [a-z] specifies a range that matches from "a" to "z" (inclusive). These forms can be mixed so [abce-g] matches "a", "b", "c", "e", "f" or "g". If the character after the [is a ! then it is used for negation so [!a-c] matches any character except "a", "b", or "c".
Within a bracket expression the *, ? and \ characters match themselves. The (-) character matches itself if it is the first character within the brackets, or the first character after the ! if negating.

The { } characters are a group of subpatterns, where the group matches if any subpattern in the group matches. The "," character is used to separate the subpatterns. Groups cannot be nested.

Leading period/dot characters in file name are treated as regular characters in match operations. For example, the "*" glob pattern matches file name ".login". The Files.isHidden method may be used to test whether a file is considered hidden.

All other characters match themselves in an implementation dependent manner. This includes characters representing any name-separators.

The matching of root components is highly implementation-dependent and is not specified.

When the syntax is "regex" then the pattern component is a regular expression as defined by the Pattern class.

For both the glob and regex syntaxes, the matching details, such as whether the matching is case sensitive, are implementation-dependent and therefore not specified.

Parameters:
syntaxAndPattern - The syntax and pattern

Returns:
A path matcher that may be used to match paths against the pattern

Throws:
IllegalArgumentException - If the parameter does not take the form: syntax:pattern PatternSyntaxException - If the pattern is invalid UnsupportedOperationException - If the pattern syntax is not known to the implementation

Complete the following code fragment so that it will print owner's name of a file:

```
Path path = Paths.get("c:\\temp\\test.txt");

//INSERT CODE HERE

System.out.println(ownername);
```

(Assume that the file system supports owner name for a file.)

Left Unanswered

You had to select 1 option

```
PosixFileAttributeView pfav = Files.getFileAttributeView(path, PosixFileAttributeView.class);
PosixFileAttributes attrs = pfav.readAttributes();
String ownername = attrs.getOwner().getName();
```

The name of the method to get the owner of a File is owner and not getOwner. Note that this method returns an instance of java.nio.file.attributeUserPrincipal.

```
PosixFileAttributeView pfav = Files.getFileAttributeView(path, PosixFileAttributeView.class);
PosixFileAttributes attrs = pfav.readAttributes();
String ownername = attrs.owner().getName();
```

```
AclFileAttributeView pfav = Files.getFileAttributeView(path, AclFileAttributeView.class);
PosixFileAttributes attrs = pfav.readAttributes();
String ownername = attrs.owner().getName();
```

```
AclFileAttributeView av = Files.getFileAttributeView(path, AclFileAttributeView.class);
AclFileAttributes attrs = av.readAttributes();
String ownername = attrs.getOwner().getName();
```

Explanation

FileOwnerAttributeView is the super class of the interfaces that supports the owner attribute. It is extended by PosixFileAttributeView and AclFileAttributeView.
Here are the details of PosixFileAttributeView:

A file attribute view that provides a view of the file attributes commonly associated with files on file systems used by operating systems that implement the Portable Operating System Interface (POSIX) family of standards.
Operating systems that implement the POSIX family of standards commonly use file systems that have a file owner, group-owner, and related access permissions. This file attribute view provides read and write access to these attributes.

The readAttributes method is used to read the file's attributes. The file owner is represented by a UserPrincipal that is the identity of the file owner for the purposes of access control. The group-owner, represented by a GroupPrincipal, is the identity of the group owner, where a group is an identity created for administrative purposes so as to determine the access rights for the members of the group.

The permissions attribute is a set of access permissions. This file attribute view provides access to the nine permission defined by the PosixFilePermission class. These nine permission bits determine the read, write, and execute access for the file owner, group, and others (others meaning identities other than the owner and members of the group). Some operating systems and file systems may provide additional permission bits but access to these other bits is not defined by this class in this release.

Usage Example: Suppose we need to print out the owner and access permissions of a file:

```
Path file = ...
PosixFileAttributes attrs = Files.getFileAttributeView(file, PosixFileAttributeView.class)
    .readAttributes();
System.out.format("%s %s%n",
    attrs.owner().getName(),
    PosixFilePermissions.toString(attrs.permissions()));
```

Add Note

What can be inserted in the code below so that it will print true when run?

```
public class TestClass{

   public static boolean checkList(List list, Predicate<List> p){
      return p.test(list);
   }

   public static void main(String[] args) {
      boolean b = //WRITE CODE HERE
      System.out.println(b);
   }
}
```

You answered incorrectly

You had to select 2 options

- [ ] `checkList(new ArrayList(), al -> al.isEmpty());`

The `test` method of `Predicate` returns a `boolean`. So all you need for your body part in your lambda expression is an expression that returns a `boolean`.
`isEmpty()` is a valid method of ArrayList, which returns true if there are no elements in the list. Therefore, al.isEmpty() constitutes a valid body for the lambda expression in this case.

- [ ] `checkList(new ArrayList(), ArrayList al -> al.isEmpty());`

You need to put the parameter list of the lambda expression in brackets if you want to use the parameter type. For example,
`checkList(new ArrayList(), (List al) -> al.isEmpty());`
Remember that specifying the parameter type is optional (as shown in option 1) because the compiler can figure out the parameter types by looking at the signature of the abstract method of any functional interface (here, Predicate's test method).

- [ ] ~~`checkList(new ArrayList(), al -> return al.size() == 0);`~~

You need to put the body withing curly braces if you want to use the return keyword. For example,
`checkList(new ArrayList(), al -> { return al.size() == 0; });`

- [x] `checkList(new ArrayList(), al -> al.add("hello"));`

The add method of `ArrayList` returns a `boolean`. Further, it returns `true` if the list is altered because of the call to add. In this case, `al.add("hello")` indeed alters the list because a new element is added to the list.

- [x] `checkList(new ArrayList(), (ArrayList al) -> al.isEmpty());`

Predicate is typed to List (not ArrayList) in the checkList method, therefore, the parameter type in the lambda expression must also be List. It cannot be ArrayList.

Add Note

```
Map<String, List<Double>> groupedValues = new HashMap<>();

public void process(String name, Double value){
    List<Double> values = groupedValues.get(name);
    if(values == null){
        values = new ArrayList<Double>();
        groupedValues.put(name, values);
    }
    values.add(value);
}
```

Which of the following implementations correctly makes use of the Java 8 functional interfaces to achieve the same?

You answered incorrectly
You had to select 1 option

- ( )
```
public void process(String name, Double value){
    groupedValues.computeIfAbsent(name, (a)->new ArrayList<Double>()).add(value);
}
```
The objective of the given code is to collect multiple values for a given key in a map. When a value for a new key is to be inserted, it needs to put a List in the map first before adding the key to the List.

computeIfAbsent is perfect for this. This methods checks if the key exists in the map. If it does, the method just returns the value associated with that key. If it doesn't, the method executes the Function, associates the value returned by that Function in the map with that key, and returns that value.

- (•)
```
public void process(String name, Double value){
    groupedValues.computeIfAbsent(name, (a, b)->new ArrayList<Double>()).add(value);
}
```
Remember that while `compute` and `computeIfPresent` take a `BiFunction` as an argument, `computeIfAbsent` takes a Function.
`(a, b)->new ArrayList<Double>()` is valid lambda expression for a `BiFunction` but will not be helpful here. This option will, therefore, not compile.

- ( )
```
public void process(String name, Double value){
    groupedValues.computeIfPresent(name, (a, b)->new ArrayList<Double>()).add(value);
}
```
This option will compile fine but will not do what is required. We want to add new ArrayList object to the map for a given key only if the mapping is not already there. This option is doing just the opposite. It will actually throw a NullPointerException when you try to add a value to key that does not already exist in the map.

- ( )
```
public void process(String name, Double value){
    groupedValues.compute(name, (a)->new ArrayList<Double>()).add(value);
}
```
This option will not compile because compute expects a BiFunction as an argument. Further, it will not do what is required in this case.

Explanation

You need to know about the three flavors of compute methods of Map:
1. `public V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). For example, to either create or append a String msg to a value mapping: `map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))`
(Method merge() is often simpler to use for such purposes.)
If the function returns null, the mapping is removed (or remains absent if initially absent). If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.
Parameters:
`key` - key with which the specified value is to be associated
`remappingFunction` - the function to compute a value
Returns:
the new value associated with the specified key, or null if none

2. `public V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`
If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null. If the function returns null no mapping is recorded. If the function itself throws an (unchecked) exception, the exception is rethrown, and no mapping is recorded. The most common usage is to construct a new object serving as an initial mapped value or memorized result, as in:
```
map.computeIfAbsent(key, k -> new Value(f(k)));
```
Or to implement a multi-value map, Map<K,Collection<V>>, supporting multiple values per key:
```
map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);
```
Parameters:
`key` - key with which the specified value is to be associated
`mappingFunction` - the function to compute a value
Returns:
the current (existing or computed) value associated with the specified key, or null if the computed value is null

3. `public V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
If the function returns null, the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.
Parameters:
`key` - key with which the specified value is to be associated remappingFunction - the function to compute a value
Returns:
the new value associated with the specified key, or null if none

Consider the following code:

```
public class Student {
    private Map<String, Integer> marksObtained = new HashMap<String, Integer>();
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    public void setMarksInSubject(String subject, Integer marks){
      // valid code to set marks for a given subject
    }
    public double getAverageMarks(){

        //1 - INSERT CODE HERE

        double sum = 0.0;
        try{
            for(Integer mark : marksObtained.values()){
                sum = sum + mark;
            }
            return sum/marksObtained.size();
        }finally{

              //2 - INSERT CODE HERE

        }
    }
}
```

What should be inserted at //1 and //2?

You answered correctly

You had to select 1 option

- `lock.lock();` and `lock.unlock();`
- `lock.readLock();` and `lock.readUnlock();`
- `lock.read();` and `lock.unlock();`
- **(correct)** `lock.readLock().lock();` and `lock.readLock().unlock();`

Explanation

From a `ReadWriteLock`, you can get one read lock (by calling `lock.readLock()`) and one write lock (by calling `lock.writeLock()`). Even if you call these methods multiple times, the same lock is returned. A read lock can be locked by multiple threads simultaneously (by calling `lock.readLock().lock()`), if the write lock is free. If the write lock is not free, a read lock cannot be locked. The write lock can be locked (by calling `lock.writeLock().lock()`) only by only one thread and only when no thread already has a read lock or the write lock. In other words, if one thread is reading, other threads can read, but no thread can write. If one thread is writing, no other thread can read or write.

Methods that do not modify the collection (i.e. the threads that just "read" a collection) should acquire a read lock and threads that modify a collection should acquire a write lock.

The benefit of this approach is that multiple reader threads can run without blocking if the write lock is free. This increases performance for read only operations. The following is the complete code that you should try to run:

```
public class Student {
    private Map<String, Integer> marksObtained = new HashMap<String, Integer>();
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    public void setMarksInSubject(String subject, Integer marks){
        lock.writeLock().lock(); //1
        try{
            marksObtained.put(subject, marks);
        }finally{
            lock.writeLock().unlock(); //2
        }
    }
    public double getAverageMarks(){
        lock.readLock().lock(); //3
        double sum = 0.0;
        try{
            for(Integer mark : marksObtained.values()){
                sum = sum + mark;
            }
            return sum/marksObtained.size();
        }finally{
            lock.readLock().unlock();//4
        }
    }
```

What will the following code print?

```
ReentrantLock rlock = new ReentrantLock();
boolean f1 = rlock.lock();
System.out.println(f1);
boolean f2 = rlock.lock();
System.out.println(f2);
```

You answered correctly

You had to select 1 option

- true
true
- true
false
- true
- **It will not compile.**

Lock.lock() returns void. Lock.tryLock() returns boolean.

Had the code been:

```
ReentrantLock rlock = new ReentrantLock();
boolean f1 = rlock.tryLock();
System.out.println(f1);
boolean f2 = rlock.tryLock();
System.out.println(f2);
```

It would have printed:

```
true
true
```

Note that ReentrantLock implements Lock.

Add Note

What will the following code fragment print?

```
Path p1 = Paths.get("c:\\personal\\.\\photos\\..\\readme.txt");
Path p2 = Paths.get("c:\\personal\\index.html");
Path p3 = p1.relativize(p2);
System.out.println(p3);
```

You answered incorrectly

You had to select 1 option

- ..\..\..\..\index.html

Observe that if you append this path to p1, you will get p2. Therefore, this is the right answer.

```
    p1 + ..\..\..\..\index.html
=>c:\\personal\\.\\photos\\..\\readme.txt + ..\..\..\..\index.html
=>c:\\personal\\.\\photos\\.. + ..\..\..\index.html
=>c:\\personal\\.\\photos + ..\..\index.html
=>c:\\personal\\. + ..\index.html
=>c:\\personal + index.html
=>c:\\personal\index.html
```

A ".." implies parent folder, therefore imagine that you are taking off one ".." from the right side of the plus sign and removing the last name of the path on the left side of the plus sign.

- ..\..\..\index.html
- ..\index.html
- c:\personal\index.html

Explanation

You need to understand how relativize works for the purpose of the exam. The basic idea of relativize is to determine a path, which, when applied to the original path will give you the path that was passed. For example, "a/c" relativize "a/b" is "../b" because "/a/c/../b" is "/a/b" Notice that "c/.." cancel out.

Please go through the following description of relativize() method, which explains how it works in more detail.

public Path relativize(Path other)
Constructs a relative path between this path and a given path. Relativization is the inverse of resolution. This method attempts to construct a relative path that when resolved against this path, yields a path that locates the same file as the given path. For example, on UNIX, if this path is "/a/b" and the given path is "/a/b/c/d" then the resulting relative path would be "c/d".

Where this path and the given path do not have a root component, then a relative path can be constructed.

A relative path cannot be constructed if only one of the paths have a root component.

Where both paths have a root component then it is implementation dependent if a relative path can be constructed.

If this path and the given path are equal then an empty path is returned.

For any two normalized paths p and q, where q does not have a root component,
p.relativize(p.resolve(q)).equals(q)

When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the same file as other is implementation dependent. For example, if this path is "/a/b" and the given path is "/a/x" then the resulting relative path may be "../x". If "b" is a symbolic link then is implementation dependent if "a/b/../x" would locate the same file as "/a/x".

Add Note

What will the following code fragment print?

```
Path p1 = Paths.get("photos\\goa");
Path p2 = Paths.get("\\index.html");
Path p3 = p1.relativize(p2);
System.out.println(p3);
```

You answered correctly

You had to select 1 option

- ..\index.html
- \index.html
- \photos\index.html
- \photos\goa\index.html
- **java.lang.IllegalArgumentException will be thrown**

Note that if one path has a root (for example, if a path starts with a // or c:) and the other does not, relativize cannot work and it will throw an IllegalArgumentException.

Explanation

You need to understand how relativize works for the purpose of the exam. The basic idea of relativize is to determine a path, which, when applied to the original path will give you the path that was passed. For example, "a/c" relativize "a/b" is "../b" because "a/c/../b" is "a/b" Notice that "c/.." cancel out.

Please go through the following description of relativize() method, which explains how it works in more detail.

public Path relativize(Path other)
Constructs a relative path between this path and a given path. Relativization is the inverse of resolution. This method attempts to construct a relative path that when resolved against this path, yields a path that locates the same file as the given path. For example, on UNIX, if this path is "/a/b" and the given path is "/a/b/c/d" then the resulting relative path would be "c/d".

Where this path and the given path do not have a root component, then a relative path can be constructed.

A relative path cannot be constructed if only one of the paths have a root component.

Where both paths have a root component then it is implementation dependent if a relative path can be constructed.

If this path and the given path are equal then an empty path is returned.

For any two normalized paths p and q, where q does not have a root component,
p.relativize(p.resolve(q)).equals(q)

When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the same file as other is implementation dependent. For example, if this path is "/a/b" and the given path is "/a/x" then the resulting relative path may be "../x". If "b" is a symbolic link then is implementation dependent if "a/b/../x" would locate the same file as "/a/x".

Add Note

Previous Next Review Discuss*

Assuming that the directory /works/ocpjp/code exists but /works/ocpjp/code/sample does not exist, what will the following code output?

```
Path d1 = Paths.get("/works");
Path d2 = d1.resolve("ocpjp/code"); //1
d1.resolve("ocpjp/code/sample");  //2
d1.toAbsolutePath(); //3
System.out.println(d1);
System.out.println(d2);
```

Left Unanswered

You had to select 1 option

- An exception at //2.
- An exception at //3.

1. The first println for d1 is straight forward.
2. d2 is created by resolving "ocpjp/code" with "/works". Since "ocpjp/code" is a relative path, d2 will contain "/works/ocpjp/code".

- \works
\works\ocpjp\code
- \works
<some path that is system dependent>

Explanation

1. Path operations don't really care about whether the path actually exists on the file system. A Path object just represents an abstract path that may or may not exist. It is only when you actually try to create or write something to the given path that its existence is required.

2. As per JavaDoc description of Path interface, its implementations are immutable and safe for use by multiple concurrent threads. This implies that the operations such as resolve and toAbsolutePath don't change the Path object itself. They return a new Path object.

In this case, lines //2 and //3 will actually produce Path objects representing "\works\ocpjp\code\sample" and "C:\works" respectively. (Assuming that C:\ is the root of the file system.)

Add Note

What will the following code fragment print when compiled and run?

```
Locale myloc = new Locale.Builder().setLanguage("hinglish").setRegion("IN").build(); //L1
ResourceBundle msgs = ResourceBundle.getBundle("mymsgs", myloc);

Enumeration<String> en = msgs.getKeys();
while(en.hasMoreElements()){
    String key = en.nextElement();
    String val = msgs.getString(key);
    System.out.println(key+"="+val);
}
```

Assume that only the following two properties files (contents of the file is shown below the name of the file) are accessible to the code.

1. mymsgs_hinglish_US.properties

```
okLabel=OK
cancelLabel=Cancel
```

2. mymsgs_hinglish_UK.properties

```
okLabel=YES
noLabel=NO
```

You answered correctly

You had to select 1 option

- It will not compile due to line L1.

There is no problem with the statement at L1. It illustrates the correct way to use a Builder object to create a Locale.

- It will not print anything.
-
```
okLabel=OK
cancelLabel=Cancel
```
-
```
okLabel=YES
noLabel=NO
```
- **It will throw an exception at run time.**

The code is trying to create a resource bundle for a specific locale i.e. `hinglish_IN`. To create this bundle, at least one of `mymsgs_hinglish_IN.properties`, `mymsgs_hinglish.properties`, and `mymsgs.properties` must be present in the classpath. Since none of these files is not available, the resource bundle cannot be created. An exception will therefore be thrown when you call `getBundle()`.

Explanation

You need to know how a ResourceBundle is prepared for a particular locale when multiple properties files are available. The following JavaDoc API decription contains all you need to know: http://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html#getBundle-java.lang.String-java.util.Locale-java.lang.ClassLoader-

You may try executing the given code and see how the values are loaded. Keep the properties files in the base folder of your classpath. For example, if your classpath contains c:\javatest\classes, then keep the properties files in c:\javatest\classes.

Add Note

Given:

```
Locale locale = new Locale("en", "US");
ResourceBundle rb = ResourceBundle.getBundle("test.MyBundle", locale);
```

Which of the following are valid lines of code?
(Assume that the ResourceBundle has the values for the given keys.)

You answered incorrectly
You had to select 2 options

- [ ] ~~`String obj = rb.getObject("key1");`~~

getObject returns Object so you need to put a cast on the returned value if you want to assign it to a Stirng.

- [ ] `Object obj = rb.getObject("key1");`
- [x] `String[] vals =  rb.getStringArray("key2");`
- [x] `Object obj = rb.getValue("key3");`

There is no `getValue` method in ResourceBundle.

- [ ] ~~`Object obj = rb.getObject(1);`~~

Keys are always Strings. So you cannot use an int to get value for a key.

Add Note

Which of the given option when inserted in the code below will make the code print true?

```
List<String> values = Arrays.asList("Alpha A", "Alpha B", "Alpha C");
//INSERT CODE HERE
System.out.println(flag);
```

You answered correctly

You had to select 1 option

- `boolean flag = values.stream().allMatch(str->str.equals("Alpha"));`
- `boolean flag = values.stream().findFirst().get().equals("Alpha");`
- `boolean flag = values.stream().findAny().get().equals("Alpha");`
- `boolean flag = values.stream().anyMatch(str->str.equals("Alpha"));`
- None of the above.

Predicate's test in options 1 and 4 and the call to equals in options 2 and 3 always return false, whichever element we apply them to. If we replaced equals() with contains() or startsWith(), they would all be correct.

Explanation

You need to know the details of the following methods of Stream interface: allMatch, noneMatch, anyMatch, findFirst, and findAny.
Please see the JavaDoc API description of these methods: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

It is important to note that all of these are short-circuiting terminal operations. This means, the given predicate will not be executed for each element of the stream if the result can be determined by testing an element in the beginning itself. For example, if you invoke predicate on the given stream, the predicate will return false for the first element and therefore there is not need for it to be executed only rest of the element.

Add Note

Given:

```
Path p1 = Paths.get("c:\\temp\\test1.txt");
Path p2  = Paths.get("c:\\temp\\test2.txt");
```

Which of the following code fragments moves the file `test1.txt` to `test2.txt`, even if `test2.txt` exists?

You answered correctly

You had to select 2 options

- [ ] `Files.move(p1, p2);`

This will throw a `java.nio.file.FileAlreadyExistsException` if the file already exists.

- [x] `Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);`

- [ ]
```
try(Files.move(p1, p2)){
}
```

`Files.move` returns a Path object (of the destination file), which is not a resource that can be closed because it does not implement `AutoCloseable` interface. So this will not compile.

- [ ]
```
try(Files.copy(p1, p2, StandardCopyOption.REPLACE_EXISTING)){
   Files.delete(p1);
}
```

This seems like a valid option but again, it will not compile for the same reason as above.

- [x]
```
Files.copy(p1, p2, StandardCopyOption.REPLACE_EXISTING);
Files.delete(p1);
```

Explanation

`Files.copy` method will copy the file `test1.txt` into `test2.txt`. If test2.txt doesn't exist, it will be created. However, `Files.isSameFile` method doesn't check the contents of the file. It is meant to check if the two path objects resolve to the same file or not. In this case, they are not, and so, it will return false.

The following is a brief JavaDoc description for both the methods:

```
public static Path copy(Path source, Path target, CopyOption... options)
                throws IOException
```

Copy a file to a target file.
This method copies a file to the target file with the options parameter specifying how the copy is performed. By default, the copy fails if the target file already exists or is a symbolic link, except if the source and target are the same file, in which case the method completes without copying the file. File attributes are not required to be copied to the target file. If symbolic links are supported, and the file is a symbolic link, then the final target of the link is copied. If the file is a directory then it creates an empty directory in the target location (entries in the directory are not copied).

The options parameter may include any of the following:

`REPLACE_EXISTING` If the target file exists, then the target file is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.

`COPY_ATTRIBUTES` Attempts to copy the file attributes associated with this file to the target file. The exact file attributes that are copied is platform and file system dependent and therefore unspecified. Minimally, the last-modified-time is copied to the target file if supported by both the source and target file store. Copying of file timestamps may result in precision loss.

`NOFOLLOW_LINKS` Symbolic links are not followed. If the file is a symbolic link, then the symbolic link itself, not the target of the link, is copied. It is implementation specific if file attributes can be copied to the new link. In other words, the `COPY_ATTRIBUTES` option may be ignored when copying a symbolic link.
An implementation of this interface may support additional implementation specific options.

Copying a file is not an atomic operation. If an `IOException` is thrown then it possible that the target file is incomplete or some of its file attributes have not been copied from the source file. When the `REPLACE_EXISTING` option is specified and the target file exists, then the target file is replaced. The check for the existence of the file and the creation of the new file may not be atomic with respect to other file system activities.

```
public static Path move(Path source,  Path target,    CopyOption... options)
                throws IOException
```

Move or rename a file to a target file.
By default, this method attempts to move the file to the target file, failing if the target file exists except if the source and target are the same file, in which case this method has no effect. If the file is a symbolic link then the symbolic link itself, not the target of the link, is moved. This method may be invoked to move an empty directory. In some implementations a directory has entries for special files or links that are created when the directory is created. In such implementations a directory is considered empty when only the special entries exist. When invoked to move a directory that is not empty then the directory is moved if it does not require moving the entries in the directory. For example, renaming a directory on the same FileStore will usually not require moving the entries in the directory. When moving a directory requires that its entries be moved then this method fails (by throwing an IOException). To move a file tree may involve copying rather than moving directories and this can be done using the copy method in conjunction with the Files.walkFileTree utility method.

The options parameter may include any of the following:

`REPLACE_EXISTING` If the target file exists, then the target file is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.
`ATOMIC_MOVE` The move is performed as an atomic file system operation and all other options are ignored. If the target file exists then it is implementation specific if the existing file is replaced or this method fails by throwing an IOException. If the move cannot be performed as an atomic file system operation then AtomicMoveNotSupportedException is thrown. This can arise, for example, when the target location is on a different FileStore and would require that the file be copied, or target location is associated with a different provider to this object.
An implementation of this interface may support additional implementation specific options.

Where the move requires that the file be copied then the last-modified-time is copied to the new file. An implementation may also attempt to copy other file attributes but is not required to fail if the file attributes cannot be copied. When the move is performed as a non-atomic operation, and a IOException is thrown, then the state of the files is not defined. The original file and the target file may both exist, the target file may be incomplete or some of its file attributes may not been copied from the original file.

Which of the following statements is true about DSYNC constant defined in StandardOpenOption enum?

You answered correctly
You had to select 1 option

- DSYNC keeps only the file metadata and not the file content synchronized with the underlying storage device.

It does exactly the reverse. DSYNC requires that only the file data gets written out synchronously.

- DSYNC keeps only the file content synchronized with the directory of the file.

This option doesn't make any sense.

- DSYNC ensures that the file content is kept de-synchronized with the underlying storage device.
- DSYNC ensures that the file content as well as file meta data is kept de-synchronized with the underlying storage device.
- **DSYNC keeps only the file content and not the file meta data synchronized with the underlying storage device.**

D in DSYNC is for data. When you open a file with this attribute, it means that every update to the file's content will be written synchronously to the underlying storage device. The meta data may still remain unsynchronized. In other words, any change in the data of the file will be written to the storage device synchronously, but any change in the meta data may be batched and written to the storage device later. Thus, it makes file operations a slower as compared to when you open the file without any option.

SYNC makes sure that both - the data and meta data are synchronized with the storage device. Thus, it makes files operations even slower than DSYNC option.

Explanation

The following is a list of constants provided by java.nio.file.StandardOpenOption:

APPEND
If the file is opened for WRITE access then bytes will be written to the end of the file rather than the beginning.

CREATE
Create a new file if it does not exist.

CREATE_NEW
Create a new file, failing if the file already exists.

DELETE_ON_CLOSE
Delete on close.

DSYNC
Requires that every update to the file's content be written synchronously to the underlying storage device.

READ
Open for read access.

SPARSE
Sparse file.

SYNC
Requires that every update to the file's content or metadata be written synchronously to the underlying storage device.

TRUNCATE_EXISTING
If the file already exists and it is opened for WRITE access, then its length is truncated to 0.

WRITE
Open for write access.

Add Note

Which of the following is a valid event type for `StandardWatchEventKinds`?

You answered correctly
You had to select 1 option

- CREATE
- CREATED
- **ENTRY_CREATE**
- ENTRY_CREATED

Explanation

When you get WatchEvents from a `WatchKey`, you can get the event type for each WatchEvent using its `kind()` method. The return type of this method is WatchEvent.Kind<T> and it returns the one of the following constants defined by class `java.nio.file.StandardWatchEventKinds`:

`static WatchEvent.Kind<Path> ENTRY_CREATE`
Directory entry created.

`static WatchEvent.Kind<Path> ENTRY_DELETE`
Directory entry deleted.

`static WatchEvent.Kind<Path>ENTRY_MODIFY`
Directory entry modified.

`static WatchEvent.Kind<Object> OVERFLOW`
A special event to indicate that events may have been lost or discarded.

The following is an example of how this works:

```
Path path = Paths.get("C:/temp");
WatchService watchService = FileSystems.getDefault().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE,
     StandardWatchEventKinds.ENTRY_MODIFY,
     StandardWatchEventKinds.ENTRY_DELETE);
WatchKey key = watchService.take(); //waits until a key is available
for (WatchEvent<?> watchEvent : key.pollEvents())
{
   Kind<?> kind = watchEvent.kind();

   System.out.println(kind);
   System.out.println(watchEvent.count());

   if (kind == StandardWatchEventKinds.ENTRY_CREATE) {
      //do something
   }
}
watchService.close();
```

Add Note

Given:

```
List<Integer> ls = Arrays.asList(10, 47, 33, 23);

Object max = //INSERT code HERE
System.out.println(max); //1
```

Which of the following options can be inserted above so that it will print the largest number in the input stream?

You answered incorrectly
You had to select 1 option

- `ls.stream().map(a->a).max();`

This will not compile because the max method requires an argument of type Comparator.
Also, `map(a->a)` is redundant here.

- `ls.stream().max(Comparator.comparing(a->a)).get();`

`Comparator.comparing` method requires a Function that takes an input and returns a `Comparable`. This `Comparable`, in turn, is used by the `comparing` method to create a `Comparator`. The `max` method uses the Comparator to compare the elements in the stream. It returns an Optional<T>.

The lambda expression a->a creates a Function that takes an Integer and returns an Integer (which is a Comparable). Here, the lambda expression does not do much but in situations where you have a class that doesn't implement Comparable and you want to compare objects of that class using a property of that class that is Comparable, this is very useful.

- (selected) `ls.stream().reduce((a, b)->a>b?a:b);`

This is actually a valid lambda expression that implements BinaryOperator but the return type of the reduce method used here is `Optional`. Therefore, it will return `Optional` object containing 9 instead of just Integer 9 and you cannot assign an object of class Optional to a variable of type int.
You need to use the reduce method that takes identity value as the first element:
`ls.stream().reduce(Integer.MIN_VALUE, (a, b)->a>b?a:b)`
This will return an `Integer` object, which can be assigned to `max`.

- All of them.

- None of them.

Explanation

The `Stream.reduce` method needs a `BinaryOperator`. This interface is meant to consume two arguments and produce one output. It is applied repeatedly on the elements in the stream until only one element is left. The first argument is used to provide an initial value to start the process. (If you don't pass this argument, a different reduce method will be invoked and that returns an Optional object.)

Add Note

Given:

```
List<Integer> ls = Arrays.asList(10, 47, 33, 23);

Object max = //INSERT code HERE
System.out.println(max); //1
```

Which of the following options can be inserted above so that it will print the largest number in the input stream?

You answered incorrectly
You had to select 1 option

- `ls.stream().map(a->a).max();`

This will not compile because the max method requires an argument of type Comparator.
Also, `map(a->a)` is redundant here.

- `ls.stream().max(Comparator.comparing(a->a)).get();`

`Comparator.comparing` method requires a Function that takes an input and returns a `Comparable`. This `Comparable`, in turn, is used by the `comparing` method to create a `Comparator`. The `max` method uses the Comparator to compare the elements in the stream. It returns an Optional<T>.

The lambda expression a->a creates a Function that takes an Integer and returns an Integer (which is a Comparable). Here, the lambda expression does not do much but in situations where you have a class that doesn't implement Comparable and you want to compare objects of that class using a property of that class that is Comparable, this is very useful.

- `ls.stream().reduce((a, b)->a>b?a:b);`

This is actually a valid lambda expression that implements BinaryOperator but the return type of the reduce method used here is `Optional`. Therefore, it will return `Optional` object containing 9 instead of just Integer 9 and you cannot assign an object of class Optional to a variable of type int.
You need to use the reduce method that takes identity value as the first element:
`ls.stream().reduce(Integer.MIN_VALUE, (a, b)->a>b?a:b)`
This will return an `Integer` object, which can be assigned to `max`.

- All of them.

- None of them.

Explanation

The `Stream.reduce` method needs a `BinaryOperator`. This interface is meant to consume two arguments and produce one output. It is applied repeatedly on the elements in the stream until only one element is left. The first argument is used to provide an initial value to start the process. (If you don't pass this argument, a different reduce method will be invoked and that returns an Optional object.)

Add Note

Given:

```
List<String> l1 = Arrays.asList("a", "b");
List<String> l2 = Arrays.asList("1", "2");
```

Which of the following lines of code will print the following values?

```
a
b
1
2
```

You answered correctly

You had to select 1 option

~~`Stream.of(l1, l2).forEach((x)->System.out.println(x));`~~

This will print :

```
[a, b]
[1, 2]
```

`Stream.of(l1, l2).flatMap((x)->Stream.of(x)).forEach((x)->System.out.println(x));`

This will print :

```
[a, b]
[1, 2]
```

`Stream.of(l1, l2).flatMap((x)->x.stream()).forEach((x)->System.out.println(x));`

The objective of flatMap is to take each element of the current stream and replace that element with elements contained in the stream returned by the Function that is passed as an argument to flatMap. It is perfect for the requirement of this question. You have a stream that contains Lists. So you need a Function object that converts a List into a Stream of elements.
Now, List does have a method named stream() that does just that. It generates a stream of its elements. Therefore, the lambda expression `x->x.stream()` can be used here to create the Function object.

`Stream.of(l1, l2).flatMap((x)->x.iterator()).forEach((x)->System.out.println(x));`

This will not compile because the lambda expression `(x)->x.iterator()` will return an Iterator but we need a Stream.

Add Note

Given:

```
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17); //1
Stream<Integer> primeStream = primes.stream(); //2

Predicate<Integer> test1 = k->k<10; //3
long count1 = primeStream.filter(test1).count();//4

Predicate<Integer> test2 = k->k>10; //5
long count2 = primeStream.filter(test2).count(); //6

System.out.println(count1+" "+count2); //7
```

Identify correct statements.

Left Unanswered

You had to select 1 option

- It will print 4 3 if line at //6 is replaced with:
  `long count2 = primeStream().reset().filter(test2).count(); //6`

There is no reset method in Stream interface. Once a terminal operation is invoked on a Stream, it is considered closed. You cannot do anything with it. You have to create a new Stream.

- It will print 4 3 if types of `count1` and `count2` are changed to int.

Stream.count() returns long. So the types of count1 and count2 are fine.

- It will print 4 3 if line //4 and //6 are replaced with:
  `long count1 = IntStream.of(primes).filter(test1).count();//4`
  and
  `long count2 = IntStream.of(primes).filter(test2).count();//6`

This change will fail to compile because `IntStream.of` does not take a `List` as argument. It takes either an `int` or a varargs parameter of type `int`.

- It will print 34 or //43// if lines 4 to 7 are replaced with:

```
primeStream.collect(Collectors.partitioningBy(test1, Collectors.counting()))
      .values().forEach(System.out::print);
```

It can print 34 or 43 due to lack of any guarantees by the type of the map returned by the collect method.

Explanation

The only problem with the given code is that a stream cannot be reused once a terminal operation has been invoked on it. Therefore, line 6 will throw `java.lang.IllegalStateException: stream has already been operated upon or closed`.

Add Note

Given:

```
Stream<Integer> strm1 = Stream.of(2, 3, 5, 7, 11, 13, 17, 19); //1
Stream<Integer> strm2 = strm1.filter(i->{ return i>5 && i<15; });  //2
strm2.forEach(System.out::print); //3
```

Which of the following options can be used to replace line at //2 without affecting the output?

Left Unanswered

You had to select 1 option

○ `Stream<Integer> strm2 = strm1.filter(i>5).filter(i<15);`

This will not compile because i>5 and i<15 are invalid lambda expressions.

○ `Stream<Integer> strm2 = strm1.parallel().filter(i->i>5).filter(i->i<15).sequential();`

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream.

Streams are created with an initial choice of sequential or parallel execution. (For example, Collection.stream() creates a sequential stream, and Collection.parallelStream() creates a parallel one.) This choice of execution mode may be modified by the BaseStream.sequential() or BaseStream.parallel() methods.

It is not documented by Oracle exactly what happens when you change the stream execution mode multiple times in a pipeline. It is not clear whether it is the last change that matters or whether operations invoked after calling () parallel can be executed in parallel and operations invoked after calling sequential() will be executed sequentially.

In either case, in the given line of code, the end result will be sequential.

○
```
Stream<Integer> strm2 = strm1.collect(
            Collectors.partitioningBy(i->{ return i>5 && i<15; })
          ).get("true").stream();
```

This is almost correct but for the fact that the keys in the Map produced by partitioningBy Collector are `Boolean` and not `String`. Therefore, `get("true")` will return `null`. It should have been `get(true)`.

○ `Stream<Integer> strm2 = strm1.map(i-> i>5?i<15?i:null:null);`

It will print `nullnullnull71113nullnull`. This is not what we want.

Add Note

Given:

```
List<Integer> ls = Arrays.asList(1, 2, 3);
```

Which of the following options will compute the sum of all Integers in the list correctly?

You answered incorrectly

You had to select 2 options

- [ ] `double sum = ls.stream().sum();`

There no sum method in Stream. There is one in IntStream and DoubleStream.

- [x] `double sum = ls.stream().reduce(0, (a, b)->a+b);`

The reduce method performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

- [ ] `double sum = ls.stream().mapToInt(x->x).sum();`

- [x]
```
double sum = 0;
ls.stream().forEach(a->{ sum=sum+a; });
```

This code is almost correct but for the fact that only final or effective final local variables can be used in a lambda expression. Here, the code is trying to use sum and sum is not final.
Effectively final means that even though it is not declared as final, it is not assigned any value anywhere else after the first assignment. The compiler determines that this variable never changes and considers it as final.

- [ ]
```
double sum = 0;
ls.stream().peek(x->{sum=sum+x;}).forEach(y->{});
```

This has the same problem as above. sum is not final or effectively final.

Explanation

It is important that you go through the JavaDoc API description of the three flavors of reduce method given here: You should read about the three flavors of reduce method given here: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Add Note

Given:

```
List<Integer> ls = Arrays.asList(1, 2, 3);
```

Which of the following options will compute the sum of all Integers in the list correctly?

You answered incorrectly

You had to select 2 options

- [ ] ~~`double sum = ls.stream().sum();`~~

There no sum method in Stream. There is one in IntStream and DoubleStream.

- [x] `double sum = ls.stream().reduce(0, (a, b)->a+b);`

The reduce method performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

- [ ] ~~`double sum = ls.stream().mapToInt(x->x).sum();`~~

- [x]
```
double sum = 0;
ls.stream().forEach(a->{ sum=sum+a; });
```

This code is almost correct but for the fact that only final or effective final local variables can be used in a lambda expression. Here, the code is trying to use sum and sum is not final.
Effectively final means that even though it is not declared as final, it is not assigned any value anywhere else after the first assignment. The compiler determines that this variable never changes and considers it as final.

- [ ]
```
double sum = 0;
ls.stream().peek(x->{sum=sum+x;}).forEach(y->{});
```

This has the same problem as above. sum is not final or effectively final.

Explanation

It is important that you go through the JavaDoc API description of the three flavors of reduce method given here: You should read about the three flavors of reduce method given here: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Add Note

Given that java.lang.String has two overloaded toUpperCase methods - toUpperCase() and toUpperCase(Locale), consider the following code:

```
String name = "bob";
String val = null;
//Insert code here
System.out.print(val);
```

Which of the following code fragments can be inserted in the above code so that it will print BOB?

You answered incorrectly
You had to select 2 options

- [ ] `Supplier<String> s = name::toUpperCase;`
  `val = s.get();`

- [ ] `Supplier<String> s = name::toUpperCase;`
  `val = s.apply();`

The name of the method in Supplier is get (not apply).

- [ ] `Function<String> f = name::toUpperCase;`
  `val = f.get();`

Function takes one argument and returns a value. So `Function<Type>` will not compile. It should actually be Function<T, R>. For example, Function<Locale, String>. Where Locale is the type of the input and String is the type of the return value.

If you want to make use of toUpperCase(Locale) method, you should do:
`Function<Locale, String> f1 = name::toUpperCase;`
You can then use it like this:
`val = f1.apply(Locale.UK);`

- [ ] `Function<String> f = name::toUpperCase;`
  `val = f.apply();`

- [ ] `Function<String, Locale> f= name::toUpperCase;`
  `val = f.apply();`

1. It should be `Function<Locale, String> f =` instead of `Function<String, Locale> f =` because you want this function to return a String (not a Locale). The type of the return value is specified at the last position.
2. Function expects an argument to be passed. Thus, it should be `val = f.apply(Locale.US);` instead of just `val = f.apply;`.

- [x] Only one of the above is correct.

Add Note

You want to print the date that represents upcoming tuesday from now even if the current day is a tuesday. Which of the following lines of code accomplishe(s) this?

You answered incorrectly
You had to select 2 options

- [ ] `System.out.println(LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`
- [x] `System.out.println(LocalDate.now().with(TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY)));`

This will return today's date if it is a tuesday, which is not what the question wants.

- [ ] `System.out.println(new LocalDate().with(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`

You cannot create a LocalDate object using its constructor because it is private.

- [ ] `System.out.println(new LocalDate().adjust(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`

adjust is not a valid method in LocalDate.

- [x] `System.out.println(TemporalAdjusters.next(DayOfWeek.TUESDAY).adjustInto(LocalDate.now()));`

Explanation

The JavaDoc description of `java.time.temporal.TemporalAdjusters` is very helpful:

Adjusters are a key tool for modifying temporal objects. They exist to externalize the process of adjustment, permitting different approaches, as per the strategy design pattern. Examples might be an adjuster that sets the date avoiding weekends, or one that sets the date to the last day of the month.
There are two equivalent ways of using a TemporalAdjuster. The first is to invoke the method on the interface directly. The second is to use Temporal.with(TemporalAdjuster):

```
// these two lines are equivalent, but the second approach is recommended
temporal = thisAdjuster.adjustInto(temporal);
temporal = temporal.with(thisAdjuster);
```

It is recommended to use the second approach, `with(TemporalAdjuster)`, as it is a lot clearer to read in code.
This class contains a standard set of adjusters, available as static methods. These include:
finding the first or last day of the month
finding the first day of next month
finding the first or last day of the year
finding the first day of next year
finding the first or last day-of-week within a month, such as "first Wednesday in June"
finding the next or previous day-of-week, such as "next Thursday"

Add Note

You want to print the date that represents upcoming tuesday from now even if the current day is a tuesday. Which of the following lines of code accomplishe(s) this?

You answered incorrectly
You had to select 2 options

- [ ] `System.out.println(LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`
- [x] `System.out.println(LocalDate.now().with(TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY)));`

This will return today's date if it is a tuesday, which is not what the question wants.

- [x] `System.out.println(new LocalDate().with(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`

You cannot create a LocalDate object using its constructor because it is private.

- [ ] `System.out.println(new LocalDate().adjust(TemporalAdjusters.next(DayOfWeek.TUESDAY)));`

adjust is not a valid method in LocalDate.

- [ ] `System.out.println(TemporalAdjusters.next(DayOfWeek.TUESDAY).adjustInto(LocalDate.now()));`

Explanation

The JavaDoc description of `java.time.temporal.TemporalAdjusters` is very helpful:

Adjusters are a key tool for modifying temporal objects. They exist to externalize the process of adjustment, permitting different approaches, as per the strategy design pattern. Examples might be an adjuster that sets the date avoiding weekends, or one that sets the date to the last day of the month.
There are two equivalent ways of using a TemporalAdjuster. The first is to invoke the method on the interface directly. The second is to use Temporal.with(TemporalAdjuster):

```
// these two lines are equivalent, but the second approach is recommended
temporal = thisAdjuster.adjustInto(temporal);
temporal = temporal.with(thisAdjuster);
```

It is recommended to use the second approach, `with(TemporalAdjuster)`, as it is a lot clearer to read in code.
This class contains a standard set of adjusters, available as static methods. These include:
finding the first or last day of the month
finding the first day of next month
finding the first or last day of the year
finding the first day of next year
finding the first or last day-of-week within a month, such as "first Wednesday in June"
finding the next or previous day-of-week, such as "next Thursday"

Add Note

What will the following code print when executed?

```
public static void main(String[] args) {
    Set<String> names = new TreeSet<String>();
    names.add("111");names.add("222");
    names.add("111");names.add("333");
    for(String name :  names){
        switch(name){
            default : System.out.print("333 ");
                break;
            case "111" : System.out.print("111 ");
            case "222" : System.out.print("222 ");
        }
    }
}
```

You answered correctly

You had to select 1 option

- (•) 111 222 222 333
- ( ) 333 333 333
- ( ) 333 333 333 333
- ( ) 111 222 111 333
- ( ) 111 111 222 333
- ( ) 111 222 333

Explanation

There are a few things this question tries to test you on:

1. A `Set` contains unique elements. If you add an element that already exists, it is ignored. Thus, in this case, the set will only contain 3 elements (and not 4) because 111 is being added twice.

2. A `TreeSet` keeps its elements sorted. So when you iterate through a `TreeSet`, you get elements in sorted order.

3. The order of case labels in a switch block doesn't matter. So even though default is the first statement in this switch block, it will not be executed when the name is 111 or 222.

4. A `break` statement causes the execution control to come out of the switch block. If you do not have a `break` after a `case` block, the control will fall through to the next `case` block. Thus, when name is 111, the control will go to `case 111` block and, after executing the case block, fall through to the next case block. Thus, it will print 111 222.

Therefore, the given code will print 111 222 (when name is 111), 222 (when name is 222) and 333 (when name is 333).

Add Note

What can be inserted in the following code so that it will print [21, 32, 43] ?

```
List<Integer> ls = Arrays.asList(11, 22, 33);
//INSERT CODE HERE
ls.replaceAll(func);
System.out.println(ls);
```

You answered correctly

You had to select 1 option

- `Function<Integer> func = x->x+10;`

This option will not compile because the objective of a `Function` is to take argument of one type and return a value of another type and so it requires a specification of two generic types (instead of just one). For example: `Function<Integer, String> func = x->""+x;` It takes an Integer and returns a String.
It is possible for a Function to return the value of the same type as the argument. For example, `Function<Integer, Integer> func = x->x+10;` will compile fine. However, it will still not work here because List's `replaceAll` method requires a `UnaryOperator`.

- **`UnaryOperator<Integer> func = x->x+10;`**

List's `replaceAll` method takes a `UnaryOperator` as argument. This option implements a UnaryOperator correctly.
The difference between a UnaryOperator and a Function is that the type of the return value of a Function can be different from the type of its argument while return type of a UnaryOperator is always the same.

- `UnaryOperator<Integer, Integer> func = x->x+10;`
- `Consumer<Integer> func = x->x+10;`
- `Operator<Integer> func = x->x+10;`

`Operator` is not a valid functional interface. There is a `UnaryOperator` and a `BinaryOperator`.

Add Note

Q 7 of 60 QID : enthuware.ocpjp.v8.2.1215

Hide Section/Toughness Mark

The following line of code has thrown `java.nio.file.FileSystemNotFoundException` when run. What might be the reason(s)?

```
Path p1 = Paths.get(new URI("file://e:/temp/records"));
```

You answered correctly
You had to select 2 options

- [x] The file system, identified by the URI, does not exist.
- [ ] The preconditions on the uri parameter do not hold.

This could be a cause for IllegalArgumentException but not for FileSystemNotFoundException.

- [x] The provider identified by the URI's scheme component is not installed.
- [ ] The security manager is installed and it denies an unspecified permission to access the file system.

This is a cause for SecurityException.

- [ ] The passed path string cannot be converted into a Path object.

Explanation

The following JavaDoc API description of Paths.get(URI) method is important for the exam. Pay close attention to the exceptions that it throws -

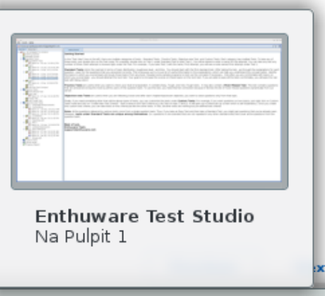
```
public static Path get(URI uri)
```
Converts the given URI to a Path object.
This method iterates over the installed providers to locate the provider that is identified by the URI scheme of the given URI. URI schemes are compared without regard to case. If the provider is found then its getPath method is invoked to convert the URI.
In the case of the default provider, identified by the URI scheme "file", the given URI has a non-empty path component, and undefined query and fragment components. Whether the authority component may be present is platform specific. The returned Path is associated with the default file system.
The default provider provides a similar round-trip guarantee to the File class. For a given Path p it is guaranteed that
`Paths.get(p.toUri()).equals( p.toAbsolutePath())`
so long as the original Path, the URI, and the new Path are all created in (possibly different invocations of) the same Java virtual machine. Whether other providers make any guarantees is provider specific and therefore unspecified.

Parameters:
uri - the URI to convert

Returns:
the resulting Path

Throws:
`IllegalArgumentException` - if preconditions on the uri parameter do not hold. The format of the URI is provider specific.
`FileSystemNotFoundException` - The file system, identified by the URI, does not exist and cannot be created automatically, or the provider identified by the URI's scheme component is not installed
`SecurityException` - if a security manager is installed and it denies an unspecified permission to access the file system

Add Note



Review Discuss*

Given the following code for monitoring a directory:

```
Path path = Paths.get(directoryPath);
WatchService watchService = FileSystems.getDefault().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE,
              StandardWatchEventKinds.ENTRY_MODIFY,
              StandardWatchEventKinds.ENTRY_DELETE);

while(true){
    WatchKey key = watchService.take(); //waits until a key is available
    System.out.println(key.isValid());

    for (WatchEvent<?> watchEvent : key.pollEvents())
    {
       Kind<?> kind = watchEvent.kind();

       System.out.println(watchEvent);
    }
}
```

A file is created and then deleted from the monitored directory. How many events will be printed by the above code?

You answered correctly
You had to select 1 option

- 0
- 1
- 2
- 3

Explanation

Once you retrieve a key from WatchService, you can't get further events until you call key.reset().
In this case, the code will get the create event but since it does not call reset() on the key before repeating the loop, it will not receive the delete event.

Add Note

What is the state of the WatchKey at the end of the following code?

```
Path path = Paths.get("C:/temp");
WatchService watchService = FileSystems.getDefault().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE,
     StandardWatchEventKinds.ENTRY_MODIFY,
     StandardWatchEventKinds.ENTRY_DELETE);
WatchKey key = watchService.take();
```

You answered incorrectly
You had to select 1 option

- Open
- Ready
- Signaled
- Closed
- Stopped

Explanation

JavaDoc API description for WatchKey says, "When initially created the key is said to be `ready`. When an event is detected then the key is `signaled` and queued so that it can be retrieved by invoking the watch service's poll or take methods. Once signalled, a key remains in this state until its reset method is invoked to return the key to the ready state."

Note: It does not explicitly mention "invalid" as one of the states but in the description for cancel() method, it says, "Cancels the registration with the watch service. Upon return the watch key will be invalid.
...
Once cancelled, a watch key remains forever invalid."

The exam has questions that require you to know valid states but it is not clear whether invalid is considered a state or not.

Add Note

You want to retrieve a WatchKey from a WatchService such that if no key is available, you want to wait for at most 1 minute. Which of the following statements will you use?

You answered incorrectly
You had to select 1 option

- `WatchKey key = watchService.take(60000);`
- `WatchKey key = watchService.poll(60000);`
- `WatchKey key = watchService.take(1, TimeUnit.MINUTES);`
- `WatchKey key = watchService.poll(1, TimeUnit.MINUTES);`

Explanation

`WatchService` has two types of methods for retrieving the WatchKeys - poll and take. The two `poll` methods return null if no key is present and do not wait for ever. The `take` method doesn't return `null` but keeps waiting until a key is available.

`WatchKey take() throws InterruptedException`
Retrieves and removes next watch key, waiting if none are yet present.
Notice that this method does not take any parameter. So it waits until a key becomes available (or until the WatchService is closed, in which case the method throws `ClosedWatchServiceException`). If you want to wait for a specified time, you should use the poll method.

`WatchKey poll(long timeout, TimeUnit unit) throws InterruptedException` : Retrieves and removes the next watch key, waiting if necessary up to the specified wait time if none are yet present.

`WatchKey poll()`: Retrieves and removes the next watch key, or null if none are present.

Add Note

A WatchKey can be in which of the following states?

You answered correctly
You had to select 2 options

- [ ] Open
- [x] Ready
- [x] Signaled
- [ ] Closed
- [ ] Stopped
- [ ] Not Ready

Explanation

JavaDoc API description for WatchKey says, "When initially created the key is said to be `ready`. When an event is detected then the key is `signaled` and queued so that it can be retrieved by invoking the watch service's poll or take methods. Once signalled, a key remains in this state until its reset method is invoked to return the key to the ready state."

Note: It does not explicitly mention "invalid" as one of the states but in the description for cancel() method, it says, "Cancels the registration with the watch service. Upon return the watch key will be invalid.
...
Once cancelled, a watch key remains forever invalid."

The exam has questions that require you to know valid states but it is not clear whether invalid is considered a state or not.

Add Note

A WatchKey can be in which of the following states?

You answered incorrectly
You had to select 3 options

- [x] Valid
- [x] Ready
- [ ] Invalid
- [x] Signaled
- [ ] Unknown

Explanation

JavaDoc API description for WatchKey says, "When initially created the key is said to be `ready`. When an event is detected then the key is `signaled` and queued so that it can be retrieved by invoking the watch service's poll or take methods. Once signalled, a key remains in this state until its reset method is invoked to return the key to the ready state."

Note: It does not explicitly mention "invalid" as one of the states but in the description for cancel() method, it says, "Cancels the registration with the watch service. Upon return the watch key will be invalid.
...
Once cancelled, a watch key remains forever invalid."

The exam has questions that require you to know valid states but it is not clear whether invalid is considered a state or not.

As per http://docs.oracle.com/javase/tutorial/essential/io/notification.html , "invalid" is a state.

Add Note

Identify the correct statements regarding the WatchService API provided by Java NIO.

You answered correctly
You had to select 2 options

- [x] You do not need to specify the `OVERFLOW` event while registering a watchable with a WatchService to receive this event.

As per the JavaDoc API description of `Watchable.register` methods, Objects are automatically registered for the `OVERFLOW` event. This event is not required to be present in the array of events.

Note that `Watchable` provides the following two methods to register for events:

```
WatchKey register(WatchService watcher, WatchEvent.Kind<?>[] events) throws IOException
```

and

```
WatchKey register(WatchService watcher, WatchEvent.Kind<?>[] events, WatchEvent.Modifier... modifiers) throws IOException
```

- [ ] You do not need to specify the `ENTRY_CREATE` or `ENTRY_DELETE` events while registering a watchable with a WatchService to receive these events.

You need to specify each event kind that you want to receive when you register a Watchable with a WatchService, except `OVERFLOW` event kind.

- [x] The counts for `ENTRY_CREATE` and `ENTRY_DELETE` are always 1.

This is as per the API description of these events. However, the count for `ENTRY_MODIFY` and `OVERFLOW` can be 1 or greater.

- [ ] The count for `ENTRY_MODIFY` and `OVERFLOW` is always 1.

For these event types, the count can be greater than 1.

- [ ] The count for any event can be greater than 1.

Add Note